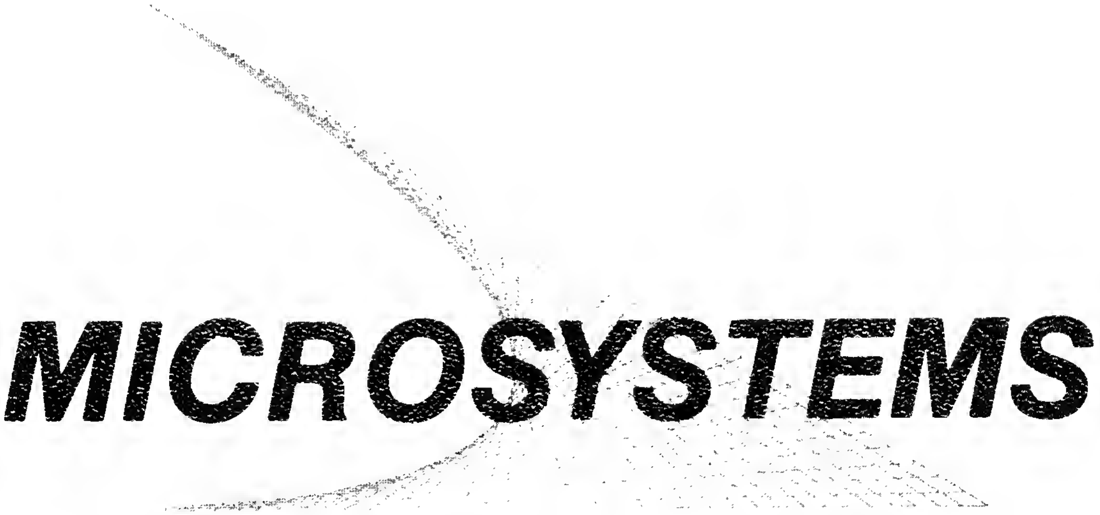
MOTOROLA

M68KDRVGD/D3

Guide to Writing Device Drivers for VERSAdos


MICROSYSTEMS

QUALITY • PEOPLE • PERFORMANCE

MOTOROLA

M68KDRVGD/D3

JANUARY 1986

GUIDE TO WRITING
DEVICE DRIVERS
FOR VERSAdos

The information in this document has been carefully checked and is believed to be entirely reliable. However, no responsibility is assumed for inaccuracies. Furthermore, Motorola reserves the right to make changes to any products herein to improve reliability, function, or design. Motorola does not assume any liability arising out of the application or use of any product or circuit described herein; neither does it convey any license under its patent rights or the rights of others.

EXORmacs, EXORterm, MACSbug, RMS68K, VERSAdos, VERSAmodule, VMEmodule, and VME/10 are trademarks of Motorola Inc.

Third Edition

Copyright 1986 by Motorola Inc.

First Edition September 1983
Second Edition March 1985

MICROSYSTEMS

REVISION RECORD

M68KDRVGD/D2 - March 1985. Reflects VERSAdos version 4.4. Adds support of the VERSAmodule 04 Monoboard Microcomputer (VM04), and VMEmodules MVME120/121 and MVME122/123. The "Driver Synthesis Example" chapter and the "Sample Driver Skeleton" appendix are removed. The driver addition algorithms for SYSGENing drivers into VERSAdos and RMS68K are revised. Chapters are added that describe the TERMLIB file, which defines device-independent routines required by driver writers, and techniques useful for debugging driver code. The appendix listing examples of CDBs and DCBs is replaced by separate appendices that describe the structure of these entities and provide updated examples. Appendices are added to describe the background and call-guarded modes of operation, and to provide examples of files created by driver writers.

M68KDRVGD/D3 -- January 1986. Minor corrections to the text have been made.

TABLE OF CONTENTS

Page

CHAPTER 1 INTRODUCTION

1.1 INTRODUCTION 1
1.2 OVERVIEW 1
1.3 RELATED PUBLICATIONS 2

CHAPTER 2 THE EXECUTIVE RMS68K

2.1 INTRODUCTION 3
2.2 THE LAYERED STRUCTURE OF RMS68K 3
2.3 THE CHANNEL MANAGEMENT REQUEST (CMR) ROUTINES 5
2.3.1 The CMR Handler Calls 7
2.3.2 Channel Types 8
2.3.3 Channel Data Blocks (CDBs) 11
2.3.4 Channel Control Blocks (CCBs) 11
2.3.5 Use of Registers Inside a Driver 12
2.3.6 Invoking the CMR Handler 12

CHAPTER 3 VERSAdos

3.1 INTRODUCTION 15
3.2 DEVICE CONTROL BLOCKS (DCBs) 15
3.3 THE IOC FILES AND MACRO FILES 15
3.4 INITIALIZATION -- GETTING THINGS STARTED 16
3.4.1 The SYSGEN System Map 16
3.4.2 The IOT Task 17
3.4.3 The IOSG Segment 18
3.4.4 The Logical Unit Table 18
3.5 COMMUNICATION BETWEEN USER TASKS AND THE I/O SYSTEM 20

CHAPTER 4 THE DRIVER

4.1 INTRODUCTION 23
4.2 INTERRUPTS AND THEIR HANDLERS 23
4.3 THE 4-ELEMENT STRUCTURE FOR A DEVICE DRIVER 25
4.3.1 The Vector Table 26
4.3.2 The Initialization Routine 27
4.3.3 The Command Service Routine 28
4.3.4 The Interrupt Service Routine 28
4.4 EQUATE FILES: IOE, TR1, TERMCCB, CCB, TCB 30
4.5 OBTAINING EXTRA MEMORY FOR THE DRIVER 30
4.6 DRIVER CALLS TO RMS68K (TRAP #0) 31

TABLE OF CONTENTS (cont'd)

Page

CHAPTER 5 I/O WITH RMS68K

5.1 INTRODUCTION 33
5.2 EXECUTION OF AN I/O REQUEST WITHOUT USING IOS 33
5.3 A DRIVER ADDITION ALGORITHM FOR SYSGENING RMS68K DRIVERS . 35
5.4 MAP OF INCLUDE FILES IN SYSGEN FILES (WITH PROCESS CONTROL DRIVER) 39

CHAPTER 6 I/O WITH VERSAdos

6.1 INTRODUCTION 41
6.2 EXECUTION OF AN I/O REQUEST USING IOS 41
6.3 A DRIVER ADDITION ALGORITHM FOR SYSGENING VERSAdos DRIVERS 41
6.4 MAP OF INCLUDE FILES IN THE SYSGEN FILES (WITHOUT PROCESS CONTROL DRIVER) 50

CHAPTER 7 TERMLIB

7.1 INTRODUCTION 51
7.2 DRIVER ROUTINE FILES 51
7.3 DRVLIB ROUTINES 51
7.3.1 QEVENT 51
7.3.1.1 QEVENT Subroutine Descriptions 52
7.3.1.2 Entry and Exit Conditions and Register Usage 52
7.3.1.3 QEVENT Examples 53
7.3.2 LOGPHY 55
7.3.2.1 Entry and Exit Conditions and Register Usage 55
7.3.2.2 LOGPHY Example 55
7.3.3 SET_TIME 56
7.3.3.1 Entry and Exit Conditions and Register Usage 56
7.3.3.2 Notes on using SET TIME 56
7.3.3.3 SET_TIME Example 56
7.4 TERMLIB ROUTINES 58
7.4.1 XDEFed Routines Called as Subroutines 58
7.4.1.1 LOG_ERR 58
7.4.1.2 RESET 59
7.4.1.3 TERM_INIT 59
7.4.1.4 TERM_COMMAND 61
7.4.1.5 TERM_BREAK 69
7.4.1.6 TERM_TBE 70
7.4.1.7 TERM_GOT CHAR 71
7.4.1.8 TERM_UNRDY 72
7.4.1.9 MARK_DOWN 73
7.4.2 Background Routines Called with the BKGRND Macro 75
7.4.2.1 How the Background and Call-Guarded Modes Work 75
7.4.2.2 Using the BKGRND and SET_BAB Macros 75
7.4.2.3 How to Write a Background Routine 77

TABLE OF CONTENTS (cont'd)

Page

7.4.2.4 RECV 77
7.4.2.5 BREAK 79
7.4.2.6 XMIT 79
7.4.2.7 BLOCK 84
7.4.2.8 UNBLK 84
7.4.2.9 STOP 85
7.4.2.10 B_BRK 85
7.4.2.11 E_BRK 85
7.4.3 Transparent Mode Routines as Implemented in TERMLIB..... 85
7.4.3.1 How Transparent Mode Is Set Up 85
7.4.3.2 TM_OUTPUT 87
7.4.3.3 TM_BREAK 87
7.5 WRITING THE DEVICE-DEPENDENT MODULE 87
7.5.1 Tables and Routines Required by TERMLIB 87
7.5.1.1 Branch Table 87
7.5.1.2 Initialization Requirements 88
7.5.1.3 PUT_CHAR: Put Out a Character to the Device 89
7.5.1.4 CK_TBE: Check to See if the Transmit Buffer Is Empty 89
7.5.1.5 DDP_RESET: Device-Dependent Reset 89
7.5.1.6 SETUP: Set Up the Device According to the Configuration 89
7.5.1.7 CLOCK_RESET: Reset the Clock 90
7.5.1.8 GET_STAT: Get Device Status 90
7.5.1.9 DDP_STOP: Device-Dependent STOP 91
7.5.1.10 DDP_UNSTOP: Device-Dependent UNSTOP 91
7.5.1.11 DDP_BEG_BREAK: Device-Dependent Begin BREAK 92
7.5.1.12 DDP_END_BREAK: Device-Dependent End BREAK 92
7.5.2 Required INCLUDE Files 92
7.5.3 An Example: The MPSC Driver Structure 92
7.6 SYSGENING THE NEW DRIVER INTO VERSAdos 97
7.6.1 TCHTYPE.AG 97
7.6.1.1 Channel Type 98
7.6.1.2 Driver Code 98
7.6.1.3 Attributes Mask 98
7.6.1.4 Parameters Mask 99
7.6.1.5 Mask of Recognized Baud Rates 99
7.6.2 MYDRIVER.CI 99
7.6.3 MYDRIVER.LG 100

CHAPTER 8 DEBUGGING THE DRIVER

8.1 INTRODUCTION 101
8.2 DRIVER DEBUGGING TECHNIQUES FOR VERSAdos 101
8.3 I/O EVENT STRUCTURE 103

TABLE OF CONTENTS (cont'd)

Page

APPENDIX A THE CHANNEL MANAGEMENT REQUEST (CMR) HANDLER 111
APPENDIX B DRIVER CALLS TO RMS68K (TRAP #0) 125
APPENDIX C VERSAdos TRAP #n MAP 157
APPENDIX D CDB STRUCTURE 159
APPENDIX E DCB STRUCTURE 175
APPENDIX F SAMPLE FILES CREATED BY THE DRIVER WRITER 181
APPENDIX G BACKGROUND AND CALL-GUARDED MODES 205

LIST OF ILLUSTRATIONS

FIGURE 2-1. Representation of Vector Chains 6
2-2. Representation of Supervisor/Subordinate Chain 10
2-3. Initiate I/O Directive Flow 13
3-1. SYSGEN System Map (EXORmacs) 16
3-2. Format of the Logical Unit Table 19
4-1. Representation of Interrupts, CCBs, and the CMR Handler .. 24
4-2. Structure of a Driver 29
5-1. Executing an I/O Request without Using IOS 34
6-1. Executing an I/O Request Using IOS 42
6-2. Overview of Driver Installation into VERSAdos 43
7-1. Background Activation Block Structure 76
7-2. SET_BAB Macro Example 76
7-3. User CSB Entering Configure-Defaults Command 86
7-4. User CSB Exiting Configure-Defaults Command 86
7-5. MPSC Driver Structure (4 Sheets) 93
7-6. TERMLIB.TF Structure 97

LIST OF TABLES

TABLE 2-1. RMS68K Functional Layers 4
7-1. IOS Command/Routine Hierarchy 63

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION

This document defines and describes the information necessary for writing an Input/Output (I/O) device driver to run under VERSAdos. An I/O device driver (also referred to as an I/O handler) is the portion of an I/O system which is responsible for the device-dependent functions corresponding to a particular device.

NOTE

> Unless otherwise specified, the designations "M68000" and "MC68000" refer to the entire M68000 family of microprocessors.

1.2 OVERVIEW

Chapter 2 describes the Real-Time Multitasking Software for the M68000, referred to as RMS68K or the Executive (EXEC) throughout this document. All data structures required by, and services provided for, the driver writer that are supported directly by the RMS68K kernel are described here.

Chapter 3 adds descriptions of those services and structures supported by VERSAdos directly to those of the EXEC.

Chapter 4 defines and describes the driver structure itself, and provides a description of supporting equate files and EXEC system calls.

Chapters 5 and 6 describe, respectively, the (I/O) sequences under RMS68K and VERSAdos, and also provide an algorithm for adding drivers to a system running either directly under the EXEC (Chapter 5) or under VERSAdos (Chapter 6).

Chapter 7 describes the TERMLIB file, which defines the device-independent routines required in writing drivers. It discusses each TERMLIB routine and also explains how to write a driver using TERMLIB and how to incorporate it into the operating system.

Chapter 8 describes techniques useful for debugging drivers, and includes a discussion of the I/O completion event structure.

Seven appendices are also included to provide in one document all the reference information a driver writer might need while developing a driver. These appendices discuss the Channel Management Request (CMR) handler; driver calls to RMS68K (TRAP #0); services provided by TRAP #n calls; the structure of Channel Data Blocks (CDBs), Channel Control Blocks (CCBs), and Device Control Blocks (DCBs); and the characteristics of background and call-guarded modes. They also provide sample CDBs, DCBs, and numerous files that are created by the driver writer.

This manual assumes that the user is already familiar with the VERSAdos utility SYSGEN and its associated files. Previous experience with SYSGEN execution and familiarity with the files IOC.xxxxDRV.AG files will be helpful.

Filenames referred to in this guide are files from either a source or object release of VERSAdos. The user numbers and catalog names of files may be found by searching the directories of the release media.

1.3 RELATED DOCUMENTATION

The following publications may provide additional helpful information. If not shipped with this product, they may be obtained from Motorola's Literature Distribution Center, 616 West 24th Street, Tempe, AZ 85282; telephone (602) 994-6561.

DOCUMENT TITLE	MOTOROLA PUBLICATION NUMBER
System Generation Facility User's Manual	M68KSYSGEN
M68000 Family Real-Time Multitasking Software User's Manual	M68KRMS68K
VERSAdos Data Management Services and Program Loader User's Manual	RMS68KIO
VERSAdos to VME Hardware and Software Configuration User's Manual	MVMEVDOS
RAD1 Device Driver Software User's Manual	M68KRADDRV
16/32-Bit Microprocessor Programmer's Reference Manual	M68000UM
SYMbug/A and DEbug Monitors Reference Manual	M68KSYMBG

CHAPTER 2

THE EXECUTIVE RMS68K

2.1 INTRODUCTION

The Motorola MC68000 Real-Time Multitasking Software is a multitasking kernel around which real-time application systems can be built. The powerful, yet general, nature of the Multitasking Software allows a wide variety of application systems to be developed without a large expenditure of design and programming effort on the complex real-time and multitasking functions.

RMS68K is compatible with the Motorola MC68000 microprocessor and the VERSAdos operating system. Therefore, programs designed to execute under the control of RMS68K will also execute under the control of VERSAdos on systems built around the MC68000.

A real-time system must respond to external events as those events occur. Because it is not known at which point an external event will occur, a real-time system is said to execute asynchronously.

Unlike a batch system where one operation is completed before a new operation is started, a real-time system can delay the completion of one operation so that another operation can be started, continued, or completed. This mechanism, where more than one operation is in progress at a given time, is called concurrent processing. Even though only one operation can be executing at a given time using a single central microprocessing unit, the concurrent processing mechanism of a real-time system makes it appear as though several operations are executing simultaneously.

A real-time application system can be broken down into several tasks. A task is a function (or operation) which can execute concurrently with other functions. A task can be written to process a single type of event, or it can process more than one type of event.

2.2 THE LAYERED STRUCTURE OF RMS68K

RMS68K is comprised of a task controller, an intertask communication facility, an optional memory management facility, and an initialization facility. These facilities allow RMS68K to perform the following functions.

- Receive all hardware and software interrupts and dispatch them to the proper task for processing.
- Dispatch tasks competing for use of the microprocessing unit.
- Provide intertask communication and synchronization.

- Manage and allocate memory.

- System initialization capability.

- Diagnostic feedback during error conditions.

RMS68K is structured in logical layers with each layer performing a particular range of functions, as described in Table 2-1.

TABLE 2-1. RMS68K Functional Layers

LAYER	DESCRIPTION
Internal	Provides functions used by RMS68K to manage the processor tasks, and physical devices such as timers. Performs work on behalf of requests from user tasks.
External	Provides functions directly available to user tasks through the use of directives.
Channel Management Request routines	Provides channel-oriented physical I/O functions. All device drivers supplied with VERSAdos and RMS68K use this optional channel management mechanism for I/O.

User tasks can request RMS68K to perform functions by using Executive directives. An Executive directive contains all of the information needed by RMS68K to perform the desired function. Functions provided are described in Chapter 4 of the M68000 Family Real-Time Multitasking Software User's Manual.

RMS68K is supplied in two parts: one part is the internal/external layer software; the other part is the CMR software. The internal/external layer software requires approximately 800 bytes of RAM memory for system parameter data, and approximately 18Kb of memory for program code. This program code can be ROM resident or bootstrap-loaded RAM. Dynamic system tables created during execution of RMS68K are not reflected in these numbers.

The CMR software requires an additional 2Kb of memory for program code. This also can be ROM resident or bootstrap-loaded RAM.

The hardware requirements for an RMS68K-based application system are:

- MC68000 microprocessor
- Adequate memory
- Optional real-time clock

The amount of memory required will vary from one system to another, depending on the system environment, user-supplied code and data, and the RMS68K functions configured in the user system. The maximum memory requirement for the entire RMS68K is provided in a preceding paragraph. This can be decreased by including only the necessary RMS68K functions in the system.

If the system is to use the delay task and periodic task activation functions of RMS68K, a real-time clock must be provided. This can be accomplished by means of a software clock mechanism that the user configures as part of RMS68K, or by means of a hardware device such as the Motorola 6840 Programmable Timer Module. The requirements listed above may be satisfied through use of a Motorola VERSAmodule Monoboard Microcomputer, VMEmodules, or a user-designed MC68000 microprocessor-based module.

2.3 THE CHANNEL MANAGEMENT REQUEST (CMR) ROUTINES

CMR routines reside as part of RMS68K. They logically manage channels and provide the link between memory mapped I/O space, interrupt vectors, and interrupt service routines. They also provide the link between requester and command service routines.

A channel is a single contiguous portion of memory mapped I/O space associated with one or more polled identity conditions of interrupt.

A channel has a corresponding hardware interrupt vector number, a hardware priority level, and a software priority number. These three items are used to link devices into polling chains, which are used by a polling routine to determine which channel is associated with an incoming interrupt. When a channel is allocated, the CMR handler creates a Channel Control Block (CCB) for that channel. The CCB contains all of the information needed by the CMR handler to manage that channel. The CCB is then placed into the appropriate polling chain. There is a polling chain for every external interrupt vector. The CCBs are chained according to the software priority number: those with higher software priority numbers will be nearer to the head of the chain, and thus serviced more rapidly when an interrupt occurs. Refer to Figure 2-1 for a representation of polling chains.

When an interrupt occurs, control is passed through the first CCB (with a Jump-to-Subroutine (JSR) instruction) to the CMR interrupt handler routine. The CMR handler will do a minimum state save, resolve the CCB address, and call the appropriate I/O handler. If the I/O handler returns without claiming the interrupt, the CMR handler will call the handler of the next CCB chained for that vector. This continues until the chain is exhausted.

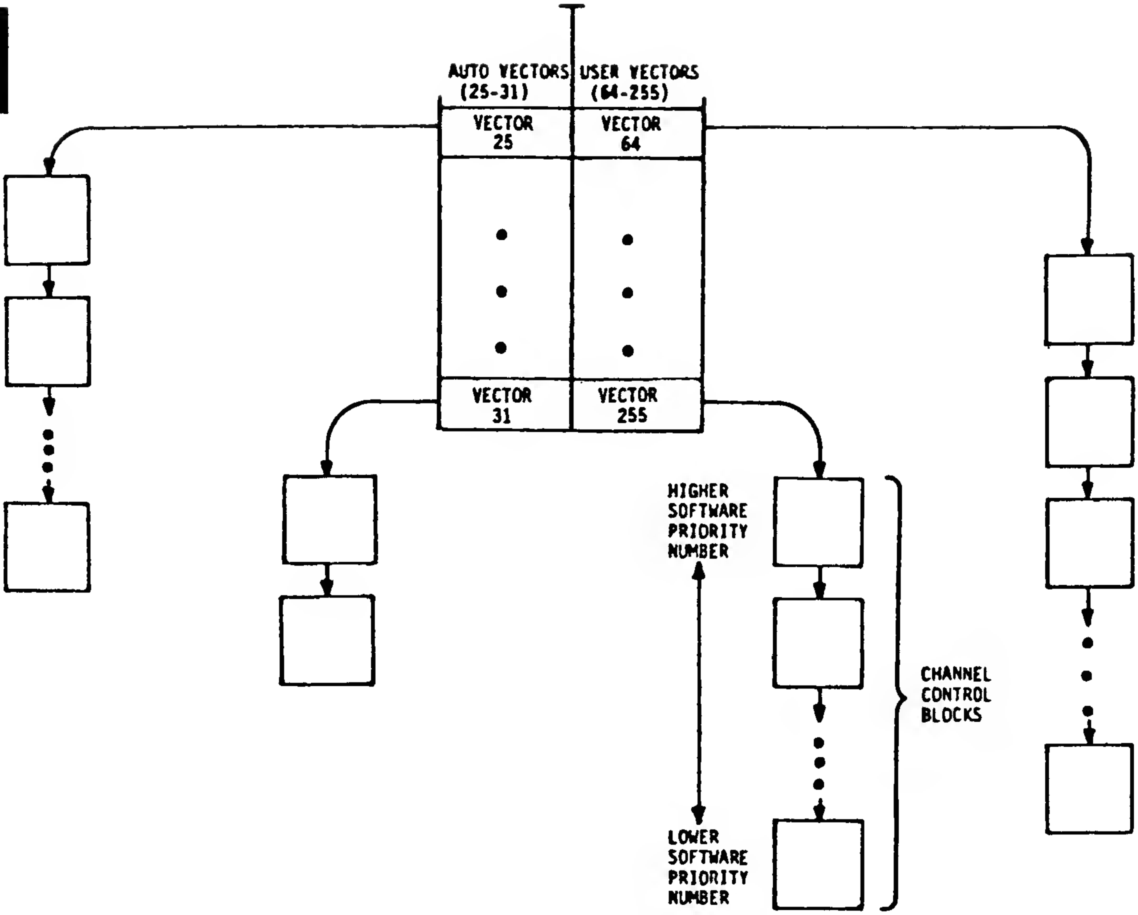

AUTO VECTORS (25-31)
USER VECTORS (64-255)
VECTOR 25
VECTOR 64
VECTOR 31
VECTOR 255
HIGHER SOFTWARE PRIORITY NUMBER
LOWER SOFTWARE PRIORITY NUMBER
CHANNEL CONTROL BLOCKS

FIGURE 2-1. Representation of Vector Chains

2.3.1 The CMR Handler Calls

User task interaction with channels is performed through requests made to the CMR handler. The following functions are available:

CMR FUNCTION	ENTERS DRIVER	FUNCTION/DESCRIPTION
ALLOCATE	YES	Generates CCB from a Channel Data Block (CDB) and links CCB into correct position in interrupt polling chain, then enters driver at its initialization routine with JSR instruction.
ATTACH	NO	Logically connects channel to requesting task; this makes requesting task the channel driver. An ATTACH request must be made before any INITIATE requests. Channel must be online (using the PUT ON LINE request) before ATTACH requests can be made.
DELETE	NO	DETACHes channel from requesting task; removes CCB from the system.
DETACH	NO	Dissolves logical connection between specified channel and requesting task.
HALT	YES	Terminates I/O in progress on a channel. HALT is applicable to nonstandard channels only. Request passed directly to driver.
INITIATE I/O	YES	Invokes the appropriate I/O handler (driver) at command service routine. The I/O function requested is passed to the driver by way of the pointer to the Input/Output Control Block (IOCB; refer to I/O Services description). An ATTACH request is required before any INITIATE requests.
PUT ON LINE	NO	Removes the specified channel from offline status, thus allowing ATTACH requests.
RESET	YES	Resets interrupts applicable to standard and shared-access channels only. Request passed directly to driver.
TAKE OFF LINE	NO	Gives offline status to the specified channel, thus preventing ATTACH requests.

Normally a channel is allocated when the system is initialized. When a user task must perform an I/O function on a particular device, it ATTACHes to the appropriate channel. The user task is then able to initiate I/O. When I/O function is complete, the user task can either DETACH from the channel or remain attached for future I/O requests. The CMR calls and parameter blocks are detailed in Appendix A.

2.3.2 Channel Types

Four types of channels are available for use under RMS68K/VERSAdos: standard, nonstandard, shared-access, and interrupt-only. Channel types are identified by a number defined when the channel is allocated by the CMR handler.

Standard channels are any channels being added to a VERSAdos operating system. The CMR handler makes a variety of checks on parameter blocks used to perform communication over standard channels.

Nonstandard channels are those that do not have the CMR handler check the parameter blocks; thus, users can define their own protocols, which would be included as part of the driver code.

Shared-access channels are those that allow more than one physical device to be attached to them, thus enabling several user tasks to use the channel simultaneously.

Interrupt-only channels are those that provide a means for external devices to interrupt the operating system but not communicate in any other way (i.e., pass no information to and accept no information from the operating system).

A code which indicates channel type is defined for a channel when that channel is allocated. Codes in the range of $10 through $7F are reserved for standard VERSAdos channels (including serial port channels). Values in the range of $01 through $0F indicate nonstandard channels. Values in the range of $80 through $8F indicate shared channels that can initiate I/O by more than one task. The value $FF indicates an interrupt-only channel.

CODE	CHANNEL DEVICE
$01-$0F	Nonstandard CMR channels
$10-$5F	Other standard channels
$60-$7F	Serial port channels
$80-$8F	Shared-access channels
$FF	Interrupt-only channel

Channels $60 through $7F are reserved for serial port modules, with the even number reserved for port A and the odd number for port B. If the module has a single port, its channel number must be even, with the odd number of the pair remaining unused. This scheme is used by the generic serial driver file TERMLIB to determine module type.

File 9993.&.TCHTYPE.AG contains a table of serial port channel numbers recognized by the generic serial port driver file TERMLIB. In TCHTYPE.AG, entries are in the following format:

```
DC.B     <channel type>
DC.B     <driver code>
DC.W     <attributes mask>
DC.W     <parameters mask>
DC.L     <mask of recognized baud rates>
```

Users who wish to use TERMLIB when writing a new serial port driver must add an entry (or two entries for two ports) to the file 9993.&.TCHTYPE.AG.

Some devices possess more than one channel (e.g., some serial controller chips). In many cases, such a device gives the same interrupt, regardless of the channel which caused it, and each of the channels must inspect the device to determine whether the interrupt belongs to that channel.

This can create a problem if reading the device's registers in some way signals the device (it might clear the interrupt, for example). To provide for this situation, CMR provides a scheme for a single channel to read the device and communicate with (and pass control to) other channels.

For such a situation, supervisor and subordinate channels must be used. A SUPERVISOR CHANNEL is allocated first (option bit 4 on the ALLOCATE command), and then the SUBORDINATE CHANNELS are allocated (option bit 3) with the supervisor's channel mnemonic in field 8. Thus, the subordinate channels receive requests to initiate I/O as usual, but all their interrupts are directed to the supervisor. The supervisor (which must be written to suit, out to the appropriate subordinate channels) then fields the interrupt and farms it out to the appropriate subordinate in whatever manner the designer desires. For example, the details of entry into the subordinate channel for proper interrupt handling are left undefined; the CMR handler only passes the interrupt to the supervisor.

The supporting data structure is a linked list starting from the supervisor channel and containing all (there is no limit) subordinates, using the longword CCBSUB in the CCB. It is expected that the supervisor will search this list for a driver corresponding to the particular interrupt type.

The supervisor channel is linked into the vector chain for the specified vector, but the subordinate channels are not. Therefore, the only way they will ever be entered for interrupt service is if the supervisor channel jumps into them. Figure 2-2 depicts the chaining of subordinate channels in relation to the vector chain.

There are some rules associated with this:

- A single channel may not be both a supervisor and a subordinate.
- A channel may not be allocated as a subordinate to a channel which is not a supervisor.
- The supervisor must be allocated before its subordinates.
- The supervisor and all its subordinates must specify the same vector.
- It is illegal to ATTACH a supervisor.

Violation of any of these rules results in the EXEC error code RTCDLGCF ($09: Request conflicts with existing table entries).

User task interactions with channels are performed through requests made to the CMR handler. The available CMR functions are listed in paragraph 2.3.1.

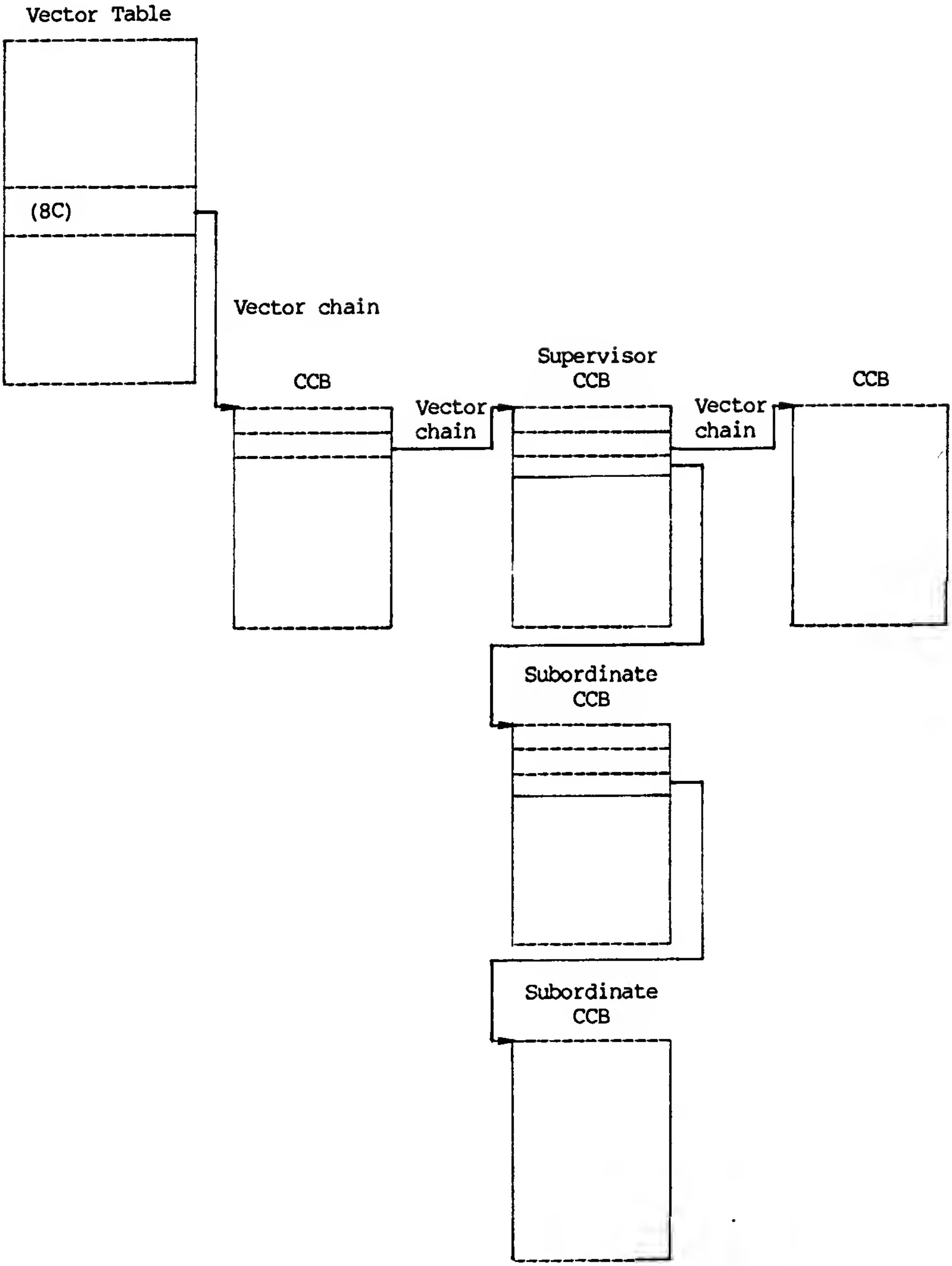

Vector Table
(8C)
Vector chain
CCB
Vector chain
Supervisor CCB
Vector chain
CCB
Subordinate CCB
Subordinate CCB

FIGURE 2-2. Representation of Supervisor/Subordinate Chain

2.3.3 Channel Data Blocks (CDBs)

A variety of options are available for CMR software. VERSAdos requires the option which enables the CMR routines. The CMR handler is responsible for establishing and managing channels. To establish a channel, an ALLOCATE call must be made to the CMR handler. The parameter block required for this call is referred to as a CDB. The result of making an ALLOCATE call to the CMR handler is a CCB, which is set up by the CMR handler from the driver writer-supplied CDB. Each channel set up requires a unique CCB; therefore, a unique CDB must be provided for each channel. Appendix D lists a standard sample VERSAdos CDB. Additional samples include the SYSGEN output files.

The file MACRO.DCB.SI contains the macro for defining the device-independent part of the DCBs. It also contains a macro for defining CDBs, the format of which is provided in Appendix D, followed by the data block field descriptions.

2.3.4 Channel Control Blocks (CCBs)

When a task (normally IOI, the VERSAdos I/O Initializer) calls upon CMR software to allocate a channel, the CMR handler dynamically creates a CCB. The CCB contains the variables that are used to control the associated channel. Because a CCB is dynamically created for each channel as the channel is allocated, I/O drivers that use CCBs should be coded as reentrant routines.

Memory is allocated in page-sized units of 256 bytes (i.e., $100) per page. Because the CMR handler allocates the memory for a CCB when a channel is allocated, a CCB always contains an integral number of pages of memory. The CMR handler always allocates at least one page of memory for a CCB because part of a page is required for universally defined variables that reside in the CCB for any type of I/O Channel. Because the CMR handler always allocates at least one page for a CCB, the first page of a CCB is referred to as the primary CCB area.

The CMR handler allocates additional page(s) for the CCB if the CMR parameter table near the beginning of the device driver for the channel specifies that additional CCB page(s) should be allocated for use by the driver. The device driver can use part of the primary CCB area and all of the additional CCB area for device-dependent variables. The additional CCB area is sometimes referred to as the extended CCB area.

The first part of a CCB contains several universally defined fields that are present in all CCBs for all kinds of channels and devices. This section deals with the universally defined fields of a CCB.

The CMR handler initializes many of the universally defined fields in a CCB at channel allocation time, and the values in these fields generally remain unchanged throughout the life of the channel. Some of the universally defined fields in a CCB are initialized by the CMR handler when a task (normally IOS, the VERSAdos Input/Output Services) attaches itself to the channel that is associated with the CCB. The values in these fields generally remain unchanged for as long as the channel remains attached to the same task. In practice, the channel always remains attached to the same task, and that task is normally IOS. A few of the universally defined variable fields in a CCB are set dynamically whenever the CMR handler is invoked to process a request to initiate an I/O transaction; these fields remain unchanged only until the next request for an I/O transaction is received. At least one universally defined field of a CCB is used by the CMR handler as a dynamic flag field, and several fields of a CCB are reserved for future use as universally defined fields. The remaining parts of a CCB are available for use by the channel's device driver for device-dependent variables.

Some of the universally defined fields in a CCB are used primarily by the CMR handler, and others are important to the I/O device driver that manages the channel. The structure of a CCB is diagrammed in Appendix D.

2.3.5 Use of Registers Inside a Driver

The registers referred to here are the MC68000 microprocessor internal address registers A0 through A7 and data registers D0 through D7. Driver writers will need to use some of these registers in their drivers. Therefore, it is important to understand which registers are available for use, and what steps must be taken if additional registers are required.

When an interrupt occurs, control is passed through the first CCB (with a JSR instruction) to the CMR interrupt handler routine. The CMR handler will do a minimum state save of registers (refer to paragraph 4.2), resolve the CCB address, and call the appropriate I/O handler (i.e., the driver). Only registers A0, A1, A5, and D0 are available for use without additional code to save the contents (as other registers require).

2.3.6 Invoking the CMR Handler

A user task makes a request to the CMR handler with an RMS68K directive. A user task issues this directive to RMS68K by executing a TRAP #1 instruction (refer to Figure 2-3). Register D0 must contain the directive number of value 60, and register A0 must contain a pointer to a parameter block. The parameter block format varies according to the exact request, and is documented in Appendix A.

An example of a user task making a CMR handler request is shown below:

```
UTSK:     .
          .
          .
          MOVE.L    #60,D0    LOAD DIRECTIVE NUMBER
          LEA       PRMBLK,A0 LOAD PARAMETER BLOCK POINTER
          TRAP      #1        ISSUE DIRECTIVE
          .
          .
          .
PRMBLK:                       DEFINE THE PARAMETER BLOCK
```

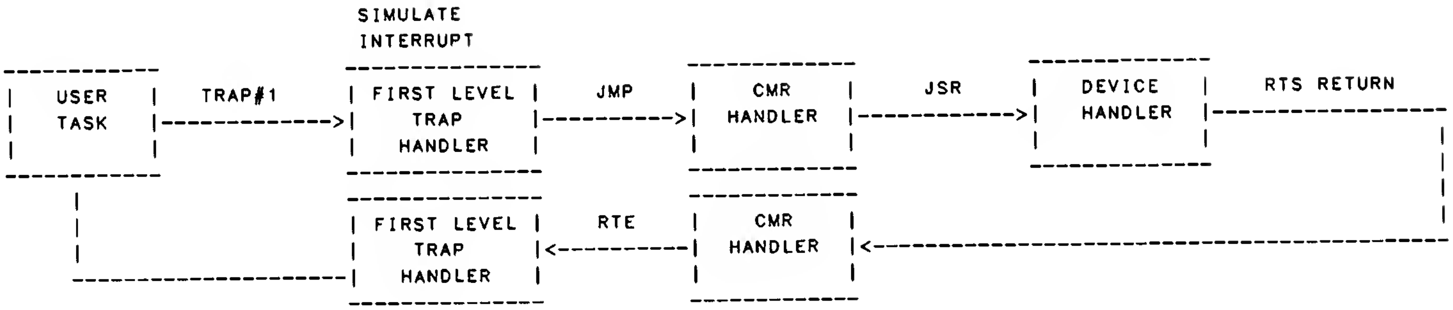

SIMULATE
INTERRUPT
USER
TASK
TRAP#1
FIRST LEVEL
TRAP
HANDLER
JMP
CMR
HANDLER
JSR
DEVICE
HANDLER
RTS RETURN
FIRST LEVEL
TRAP
HANDLER
RTE
CMR
HANDLER

FIGURE 2-3. Initiate I/O Directive Flow

RMS68K will invoke the CMR handler when a directive number 60 is issued. The CMR handler then interprets the contents of the parameter block to determine the actions to be taken. When the CMR handler receives an INITIATE I/O request, it will invoke one of the I/O handlers. The particular I/O handler invoked depends upon the service address supplied when the channel was allocated. There is one I/O handler for each channel.

When a user task issues a request to the CMR handler, the task is placed in the READY state list. When that task regains control, the CMR handler will have acted on the request. The low order byte of register D0 will contain zero if the request was successfully completed; it will contain an error code if the request was not successfully completed. The Z bit of the status register will reflect the contents of register D0: Z = 1 if register D0 = 0; Z = 0 otherwise. The value of the Z bit can be tested by using an MC68000 Bcc instruction (BEQ, BNE).

It is important to realize that if an INITIATE I/O request is made to the CMR handler, the error code returned by the CMR handler merely indicates the successful or unsuccessful invocation of the I/O handler. It does not reflect the successful or unsuccessful completion of the I/O function performed by the I/O handler.

A more complete description of CMR software and its parameter blocks is provided in Appendix A.

CHAPTER 3

VERSAdos

3.1 INTRODUCTION

VERSAdos is a modular, multilayered operating system whose structure and services are oriented to solving the general purpose software generation requirement associated with the development of microprocessor-based systems and to providing the executive requirements of dedicated, real-time, multitasking/multi-user application systems.

To illustrate the structure of VERSAdos, the following descriptions and discussions relate to Figures 3-1 and 3-2.

3.2 DEVICE CONTROL BLOCKS (DCBs)

Each channel set up in the system requires a Channel Data Block (CDB). Each device in the system may require a DCB. Only devices that are assigned using TRAP #3 and accessed through the Input/Output Services (IOS) TRAP #2 interface have DCBs. None of the 600-series process control drivers have DCBs, for example.

CDBs are discussed in Chapter 2.

DCBs are VERSAdos structures that contain information about devices (e.g., their default configuration). Devices that require DCBs include terminals, printers, disks, magnetic tape, and some special devices such as the MVME300 General Purpose Interface Bus (GPIB) module. After VERSAdos has been booted and initialized, the DCBs can be found in the shared memory segment, IOSG.

There is always a null device whose DCB is last in the chain because its pointer to the next DCB is set to zero.

Appendix E of this document and Appendix B of the System Generation Facility User's Manual provide examples of DCBs for terminals, printers, and disks.

3.3 THE IOC FILES AND MACRO FILES

To add DCBs to the operating system, they should be included in the file IOC.xxxxDRV.AG for the driver, where xxxxDRV is the name of the driver, and xxxx indicates the module number when applicable. For example, IOC.M435DRV.AG is the configuration file for the magnetic tape driver on the MVME435 module.

For a more detailed description of the IOC.xxxxDRV.AG files, refer to paragraph 6.3, "A Driver Addition Algorithm for SYSGENing VERSAdos".

Macros that define DCBs consist of two parts. The first part is the device-independent section and is found in MACRO.DCB.SI. The second part is the device-dependent section and is found in one of the following files:

MACRO.DCBDISK.SI	for disk drivers
MACRO.DCBTERM.SI	for terminal drivers
MACRO.DCBPRT.SI	for printer drivers
MACRO.DCBMTAPE.SI	for magnetic tape drivers

For a special device that does not fit in any of these categories, a new macro must be defined. For example, a new macro was defined as follows when the MVME300 (GPIB) driver was created:

MACRO.DCBGPIB.SI	for the MVME300 GPIB driver

3.4 INITIALIZATION -- GETTING THINGS STARTED

The following paragraphs describe features with which users should be familiar before using VERSAdos.

3.4.1 The SYSGEN System Map

The last thing in the SYSGEN print file is a map of the generated system. It shows the various tasks and processes present in the system when booting is complete, and the initial address where execution is to start. Refer to Figure 3-1 for the following discussion.

```
FILENAME          TASK    PROC    SEG     ADDR       ICB

RMS.LO                    RMS     RMS0   $000000
                                  RMS2   $000C00
DRVLIB.LO                 DRVL    DRVL   $005A00
TERMLIB.LO                TERM    TERM   $005C00
ACIADRV.LO                ACIA    ACIA   $006F00
PIADRV.LO                 PIAD    PIAD   $007200
IPCDRV.LO                 IPCD    IPCD   $007800
VM22DRV.LO                VM22    VM22   $008C00
FHS.LO            .FHS            .FHS   $009400   $00A800
IOS.LO            .IOS            .IOS   $00AA00   $00C400
FMS.LO            .FMS            .FMS   $00C600   $011800
EET.LO            &EET            .EET   $011A00   $015E00
                                  .STT   $011D00
                                  &EET   $012000
LDR.LO            &LDR            &LDR   $016000   $017500
IOI.LO            .IOI            IOSG   $017700   $01CE00
                                  .IOI   $01C800
SYSINIT.LO                SYSI    .INT   $01D000


-  FINAL PC VALUE = $01DA00
-  START-UP ADDRESS = $01D000
TOTAL NUMBER OF USER-DEFINED SYMBOLS = 232

0  ERRORS ENCOUNTERED
```

Figure 3-1. SYSGEN System Map (EXORmacs)

The column TASK shows the tasks in the system at the time it is booted. The TCB column shows the address in memory of the TCB for the task. A task's state and priority after the system is booted are controlled by the SYSGEN STATE and PRIORITY parameters (refer to the System Generation Facility User's Manual).

Each task can be made up of one to four segments, whose names are shown in the SEG column and whose addresses are shown in the ADDR column. The segment names are those given by the linkage editor when building the load module used to create the task.

Some load modules are used to create processes rather than tasks. A process runs in supervisor mode and, therefore, has access to all system memory and services, whereas a task runs in user mode and can access only its own declared memory space. A process does not have a TCB. Seven processes are shown: RMS, DRVL, TERM, ACIA, PIAD, IPCD, VM22, and SYSI. RMS is RMS68K, which is entered by exception processing. DRVL and TERM consist of routines common to various drivers. ACIA, PIAD, IPCD, and VM22 are all I/O device drivers. SYSI is system initialization, entered as soon as the entire system is booted. Note that the startup address is the beginning of SYSI. When SYSI is completed, it JUMPs to the dispatcher to start system processing. The memory used by SYSI is not allocated to any task, and so becomes available for system use as soon as SYSI completes.

The addresses shown in the ADDR column are the actual addresses into which the system is loaded as long as WHERLOAD is set to zero. For a VM01 system, WHERLOAD is set to load the system into offboard RAM. However, the first thing INT does is relocate the loaded system into onboard RAM, making the addresses match. INT will then continue in the relocated code.

3.4.2 The IOI Task

When SYSI completes, and turns control over to the dispatcher, the highest priority ready task is ".IOI". Entry is into module IOI (part of segment .IOI). IOI sets the I/O system in motion. Its processing can be broken down into the following four steps. For a more detailed description of the boot sequence, refer to the VERSAdos to VME Hardware and Software Configuration User's Manual.

a. Allocate all channels. This uses the CDBs assembled in SECTION 1 of the assembly of IOC.xxxxDRV.AG files and included as part of the .IOI segment. The macro "CDB" generates the CDBs. CDBs are needed only during I/O initialization, where they are converted into Channel Control Blocks (CCBs) by the CMR handler when the channels are assigned. The allocate, if successful, causes the driver to be entered at its initialization routine.

b. Calculate (and save) various data lengths. These include the data segment for FMS for its stack; Volume Device Tables (VDTs), File Control Blocks (FCBs), and File Access Tables (FATs); and ASQs for PRT, IPC, FMS, and COM.

c. Declare segment IOSG shareable.

   1. Grant shared access of segment IOSG to each VERSAdos system task.
   2. Start each task (make ready to run).

d. Terminate task .IOI. Segment .IOI disappears with the task termination, but IOSG remains because it has been made shareable and is now being used by the various I/O tasks. These tasks normally include .IOS, .FHS, and .FMS.

3.4.3 The IOSG Segment

IOSG is section 0 of the assembly of IOC.xxxxDRV.AG files to form the I/O segment. It starts out as part of task .IOI, but is given to all the I/O tasks (as described in the previous subsection) before .IOI completes. At label IOCOMS in the assembly is a table of pointers and values. This is at the beginning of the module. This table is called the System Value Table (SVT). Two sets of three pointers are of interest here:

OFFSET	POINTER
SVT + $10	Start of Logical Unit Table (LUT) space
$14	End of LUT space
$18	First LUT in chain of active LUTs
$1C	Start of DCB space
$20	End of DCB space
$24	First DCB in chain of active DCBs

To verify that the memory location is correct, users can check (using the SYSANAL utility) to make sure offset $2C contains "VERSADOSREV".

3.4.4 The Logical Unit Table

Each task has an associated LUT. Active LUTs are in a chain whose head is at the SVT+$18. SYSGEN reserves enough space for the LUT for each task (NOTASKS), with each table having room for information about one more than the maximum number of logical units available to each task (MAXLU). Logical unit 0 is reserved for system use.

Each LUT consists of a 16-byte header, followed by multiple 8-byte entries. Each 8-byte entry corresponds to one possible logical unit. The format of the LUT is shown in Figure 3-2.

The LUT for a particular session and task can be found by following the chain of active LUTs, starting with the first one (pointed to by SVT+$18) and continuing using the link pointer at offset $0 in the LUT. Users should look for the proper taskname and session number. When they are located, the entries for each logical unit can be examined. Unassigned logical units will contain zero in the LUTDCB field. Also, bit LUSFAC will be zero. Other active entries can be checked for current status in byte LUTCSF.

Bit LUSFDV indicates whether the LUT represents a file assignment or a device assignment. If it is on, it represents a device assignment. The field LUTDCB will point to a DCB for a device assignment and to an FCB for a file assignment.

Symbol	Offset		Length	Field
LUTPTR	0	($0)	4	Pointer to next table
LUTTID	4	($4)	4	Taskname
LUTSES	8	($8)	4	Task session
LUTMLU	12	($C)	1	Maximum number of LU entries
LUTCAS	13	($D)	1	Number of current assignments
LUTUNM	14	($E)	2	User number
LUTBEG	16	($10)	0	Start of LU entries
				LU entry first
LUTCAP	16	($10)	1	Current access permission
LUTCSF	17	($10)	1	Current status flag
LUTATT	18	($11)	2	Attributes of device/file
LUTDCB	20	(12)	4	Address of connected DCB/FCB
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

- Current access permission (LUTCAP)

Symbol	Value	Meaning
FOPPR	0	Public read
FOPER	1	Exclusive read
FOPPW	2	Public write
FOPEW	3	Exclusive write
FOPPRPW	4	Public read, public write
FOPPREW	5	Public read, exclusive write
FOPERPW	6	Exclusive read, public write
FOPEREW	7	Exclusive read, exclusive write

- Current status flag (LUTCSF)

Symbol	Bit	Meaning
LUSFAC	0	Active LU entry
LUSFIO	1	I/O pending
LUSFCP	2	Close pending
LUSFAS	3	Assign pending
LUSFCW	4	Connection wait
LUSFDV	7	Device assignment

- Attributes of device/file (LUTAAT) - same as DCBATT

FIGURE 3-2. Format of the Logical Unit Table

3.5 COMMUNICATION BETWEEN USER TASKS AND THE I/O SYSTEM

User tasks request service from the I/O system by using either a TRAP #2 or a TRAP #3. TRAP #3 is used to request service from File Handling Services (FHS), which runs as task .FHS. FHS handles file and device manipulation, such as allocation, assignment, and renaming. Refer to the VERSAdos Data Management Services and Program Loader User's Guide for details on the types of requests available.

TRAP #2 is used for requesting I/O operations on files or devices. These requests are handled by Input/Output Services (IOS), which runs as task .IOS. Refer to the same manual for details on types of requests available.

FHS and IOS are passed these two TRAPs because the tasks declare themselves servers of the TRAPs when they start execution. After initialization, all execution in FHS and IOS is in their Asynchronous Service Routines (ASRs). ASR is entered when issuance of the appropriate trap causes RMS68K to place user/server event (code 7) on the task's Asynchronous Service Queue (ASQ).

In addition, if there are more I/O requests pending for the same device, the device driver is rescheduled by queueing an event for the first waiting request.

At boot time, the contents of the specified boot file are loaded into memory from contiguous disk sectors, and control is transferred to the RMS68K initialization routine SYSINIT. The EXEC determines the end of contiguous physical memory for each defined partition, and initializes the free memory list to contain all memory in partition 0 excluding that occupied by the hardware vector table, all I/O driver processes, and the EXEC itself. All memory assigned to partitions other than zero is also included in the free memory list. TCBs are created and memory segments (code only) are allocated for each system task defined by SYSGEN. The standard tasks created are .FHS, .FMS, .IOS, &LDR, &EET, and .IOI. The function of each of these tasks will be described later. After completing initialization, the EXEC starts execution of .IOI. The tasks &LDR and &EET are placed on the ready list.

The function of .IOI is to initialize the I/O system. The three tasks which comprise the I/O system are .FMS, .FHS, and .IOS. The necessary data segments and ASQs are allocated by .IOI, all defined I/O channels are allocated, the I/O system tasks are started, and .IOI terminates, causing all memory allocated to its code and data segments to be returned to the free memory list.

When .IOS begins execution, it attempts to issue a Configure request to the driver of each device to initialize the current device configuration to match the default device configuration. If the Configure request results in an error, the default configuration, defined by SYSGEN, is assumed to be unreliable and the DCB for the device is marked offline, preventing any I/O from being performed on the device. This action prevents the operation of other devices from being corrupted by the actions of a driver operating on faulty configuration data. After completing internal initialization, .IOS makes the necessary EXEC calls to establish itself as a system server task for TRAP #2 and enters wait-for-event state.

When .FMS begins execution, it attempts to read sector 0 of each disk device to determine whether it is a VERSAdos disk, a foreign disk, or offline. The volume name of each VERSAdos disk is entered in an internal table, and .FMS calls .FHS to make an exclusive read/write assignment on the disk device corresponding to each volume. .FMS then enters the wait-for-event state. The .FMS initialization process will be described in more detail later.

When .FHS and &LDR begin execution, they each perform internal initialization and make calls to the EXEC to establish themselves as system server tasks for TRAP #3 and TRAP #4, respectively. Both tasks then enter wait-for-event state. The initialization performed by &EET when it begins execution will be described later.

The only element of the device I/O mechanism not described thus far is the I/O driver process. A process is defined as an extension of the EXEC that operates in the supervisor mode of the MC68000. Running in supervisor mode, the driver is not subject to preemption or time-slicing as a task would be. Another advantage is that the memory management unit (if any) is disabled in supervisor mode so that the driver has direct access to the address spaces of all tasks, system tables, and I/O devices. However, because the driver, by nature, suspends execution of all tasks when it is executing, its execution time must be kept to a minimum or the system throughput will be severely degraded. Each device driver has three major components: initialization, command service, and interrupt service.

The driver initialization routine is called when an Allocate Channel (TRAP #1) request is made to the CMR handler. The information passed in the parameter block for this call includes the hardware interrupt level, hardware interrupt vector number, and a software priority. The CMR handler uses this information to create a CCB. The device-independent portion of the CCB is initialized by the CMR handler. Before allocating the CCB, the CMR handler makes sure, by attempting to read the physical byte address specified in the CDB, that the memory mapped I/O addresses specified for the channel can be accessed. If not, an error is returned to .IOS. Each CCB is part of an interrupt vector chain. The start of each chain is the address stored in the CCB-specified hardware interrupt vector number. This address points to an instruction within the CCB of the highest software priority of all the CCBs with the same vector number. Other CCBs for that vector number are added in order of decreasing software priority to form a linked list with the first CCB in the chain. Then the driver initialization routine executes. Its function is to perform hardware initialization of the device and to set up housekeeping in the device-dependent portion of the CCB. After this is done, the driver returns to the CMR handler.

The driver command service routine is called when an Initiate I/O (TRAP #1) request is issued by .IOS. As mentioned earlier, one of the parameters passed to the CMR handler and, subsequently, to the driver is the address of the IOCB. All validation of data contained in the IOCB (excluding logical-unit-related data) is the responsibility of the command routine.

One of these checks is to verify that the specified I/O buffer starting and ending addresses are contained within the address space of the proper task. In most cases, the proper task is the owner of the IOCB (i.e., the task that issued the TRAP #2); but this is not the case if IOCB option bit 15 is set. This option bit (marked as reserved in user documentation) extends the normal IOCB by eight bytes, which contain the task and session number of the buffer owner. Use of this option is necessarily restricted to system tasks because a more general usage would allow one task to read or write to another task's address space without prior consent. (Note: This option is currently used by .FMS to make disk I/O requests on behalf of a task that requested file I/O, causing data to be transferred directly to or from the second task's buffer.) If IOCB errors are detected, the appropriate error status is written in the IOCB, and the driver returns to the CMR handler. If no errors were detected in the IOCB, the driver initiates the I/O on the device, enables I/O completion interrupts, and returns to the CMR handler. The command service routine should perform all non-interrupt sensitive operations with the processor mask set to zero.

CHAPTER 4

THE DRIVER

4.1 INTRODUCTION

It is the intention within an operating system to keep all references to I/O devices logical rather than physical for as long as possible. When the time comes to make the transition from logical to physical, I/O device drivers are required. I/O device drivers are the software modules that actually handle the physical I/O device. It is at this level that the operating system has to take into account the peculiarities of the peripheral chips. This chapter details the structure and nature of I/O device drivers (i.e., handlers) that must be observed in order to run under RMS68K and/or VERSAdos.

4.2 INTERRUPTS AND THEIR HANDLERS

What happens when an interrupt occurs? Answering this question requires knowledge of the relevant data structures as they have been set up for use by factors such as system initialization.

Considering the CDB first, there are three pertinent parameters:

- the channel driver's physical address
- the hardware interrupt vector number
- the software priority

When the I/O Initializer (IOI) runs, it calls the CMR handler to create Channel Control Blocks (CCBs) from Channel Data Blocks (CDBs); it collects all CCBs having the same hardware interrupt vector number and orders them into a chain based on software priority. The CCB with the highest software priority is placed first in the chain and so on down to the lowest software priority. At this point, there are several ordered chains of CCBs: one chain for each of the hardware interrupt vector numbers used in the particular system (auto vectors or user vectors). IOI also places in the MC68000 vector table, at the position of each hardware interrupt vector, the address of the middle of the CCB occurring at the head of the CCB chain for that hardware interrupt vector.

CMR places in the middle of each CCB a Jump-to-Subroutine (JSR) instruction to the CMR interrupt handler, with the JSR at the head of each CCB chain having its JSR to the CMR interrupt handler pointed to by the entry in the MC68000 vector table. Figure 4-1 diagrams CCB chaining. It should be noted at this point that any of the CCBs shown in Figure 4-1 could be a supervisor CCB. In this case, the supervisor CCB would point to the first of its subordinate CCBs which would point to yet further subordinate CCBs. Refer to the discussion on supervisor and subordinate CCBs which appears in Chapter 2 ("Channel Types").

When a hardware interrupt occurs in an MC68000-based system, the MC68000 Program Counter (PC) is pushed onto the supervisor stack and loaded in its place with the location found in the vector table for that interrupt. The status register is also pushed onto the supervisor stack.

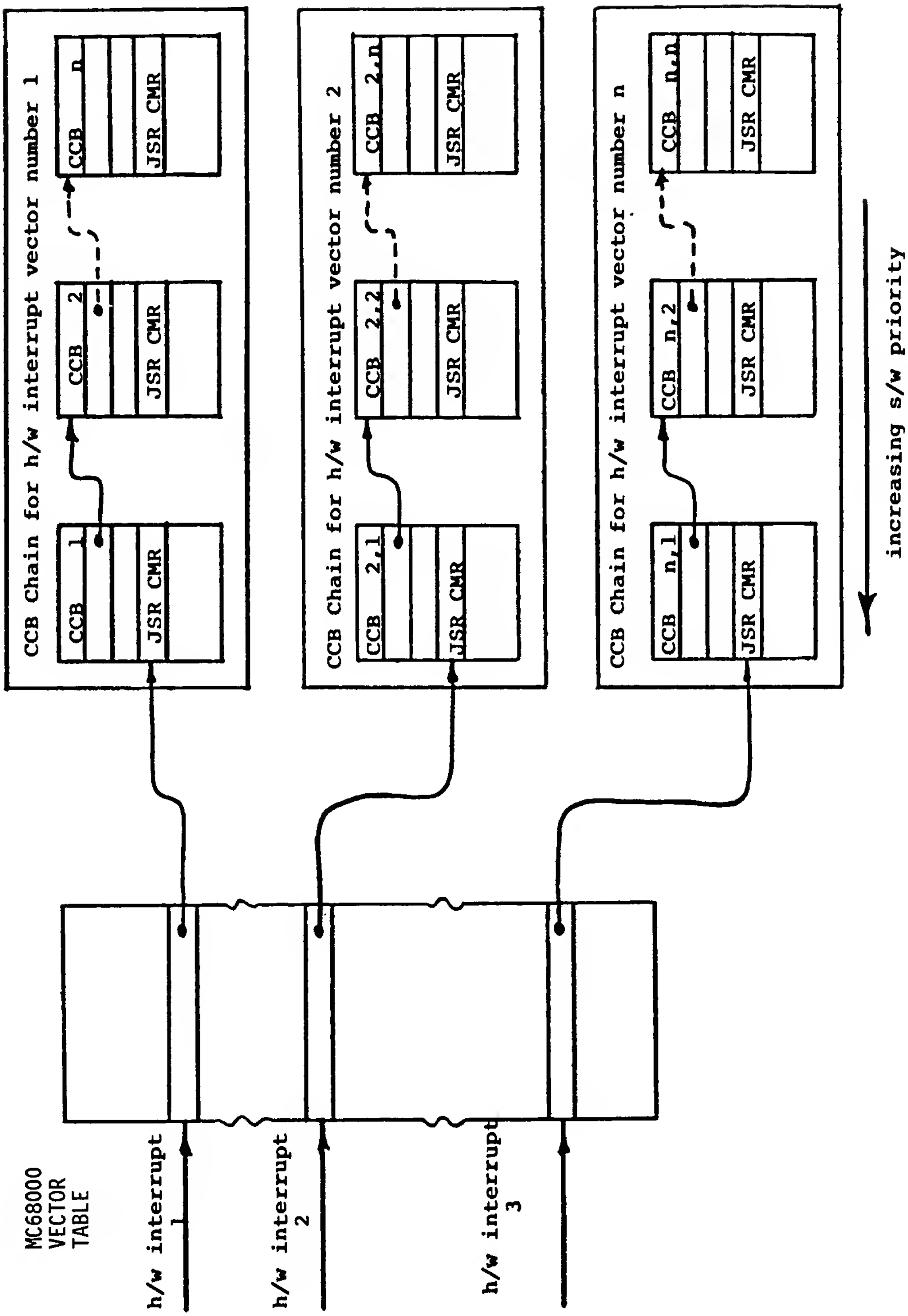

MC68000 VECTOR TABLE
h/w interrupt 1
h/w interrupt 2
h/w interrupt 3
CCB Chain for h/w interrupt vector number 1
CCB 1
JSR CMR
CCB 2
JSR CMR
CCB n
JSR CMR
CCB Chain for h/w interrupt vector number 2
CCB 2,1
JSR CMR
CCB 2,2
JSR CMR
CCB 2,n
JSR CMR
CCB Chain for h/w interrupt vector number n
CCB n,1
JSR CMR
CCB n,2
JSR CMR
CCB n,n
JSR CMR
increasing s/w priority

FIGURE 4-1. Representation of Interrupts, CCBs, and the CMR Handler

This new PC value points into the middle of the first CCB on the chain for that interrupt vector. The JSR instruction found at this position in the CCB transfers control to the CMR interrupt handler. The CMR interrupt handler traces the interrupt if interrupt tracing is on, and then performs a minimal MC68000 state save consisting of A0, A1, A5, and D0. The CMR interrupt handler pulls the return address off the stack and uses this address to calculate the address of the CCB responsible for initially calling it. It gets the address of the driver associated with that CCB (from the CCB itself) and does a JSR at the interrupt entry point of the driver, with A5 now pointing to the CCB.

The driver's first task is to determine if the device it controls is responsible for the interrupt. If this driver's device is responsible, the interrupt is handled. When the driver is done, it sets the carry bit and returns to the CMR handler. If, however, the interrupt was not from this driver's device, the carry bit is cleared and a return to the CMR handler is issued. The CMR handler then checks the carry bit. If it is clear, the interrupt has not been handled. The CMR handler must then walk one link in the CCB chain if not already at its end, and again enter the CMR interrupt handler, but from the next CCB. Walking the CCB chain continues until the interrupt is handled or the CMR handler finds itself at the end of the chain.

Either way, the CMR handler restores the saved state and exits through the common interrupt handler. The common interrupt handler deals with the interrupt in one of two ways:

a. If no interrupt was pending before this one, the handler jumps to the dispatcher.

b. If an interrupt was pending prior to this one, the handler will service it.

The faster a device's interrupt must be handled, the higher software priority required by the CCB for that device.

4.3 THE 4-ELEMENT STRUCTURE FOR A DEVICE DRIVER

To write and incorporate a driver into RMS68K/VERSAdos in the standard way, it must conform to the standard driver format, which consists of four elements. These four elements, listed below, are described in the following paragraphs.

- Vector table and revision information
- Initialization routines
- Command service routines
- Interrupt service routines

4.3.1 The Vector Table

The first 16 bytes of the device driver comprise the driver vector table. The first 4-byte vector must be the address of the Driver's Interrupt Service Routine (DISR), the second 4-byte vector that of the Driver's Command Service Routine (DCSR), the third 4-byte vector that of the Driver's Initialization Routine (DIR), and the final 4-byte vector is reserved. A further 16 bytes offer space for other CMR software parameters. Only the first byte is currently in use; it is used to increase the size of pages used for the CCB. These extra pages can be used as driver scratchpad RAM. Driver revision information can be included after the initial 32 bytes. This revision information is not required by the CMR handler, although its inclusion is recommended in case there is some confusion regarding the revision of a particular driver being used in the system. The layout of the vector table follows.

```
*****************************************************************************
*  This is the service vector table used by the Channel Management Request
*  (CMR) handler to calculate entry points to the driver.  The three entry
*  points are as follows:
*
*          1. Interrupt      - The CMR handler calls the driver at this entry
*                              point for each CCB in a specific hardware vector
*                              chain that is serviced by the driver until the
*                              interrupt is claimed.
*
*          2. Command        - Called by the CMR handler whenever an INITIATE
*                              I/O channel command is received from IOS or
*                              another task. After an I/O operation is started,
*                              control is usually returned to the CMR handler
*                              pending receipt of an interrupt from the device.
*
*          3. Initialization - Called by the CMR handler whenever a channel is
*                              allocated to allow the driver to initialize any
*                              device or devices associated with that channel.
*
*  The service vector table must reside in the first 16 bytes of the driver
*  starting with the SYSGEN-defined driver origin. The vectors must be long
*  words in the order of interrupt, command, initialization, and reserved.
*  The vectors are specified as offsets relative to the driver origin address
*  to allow relocation of the driver without reassembly or relinking.

*****************************************************************************
```

```
DEVDRVR:
         DC.L     DEVISR-DEVDRVR  Self-relative interrupt service address
         DC.L     DEVUSER-DEVDRVR Self-relative command service address
         DC.L     DEVINIT-DEVDRVR Self-relative initialization service address
         DC.L     0               Reserved
*
*  The following 16 bytes are reserved for use as channel allocation
*  parameters by the CMR handler.  Only the first byte is currently used.
*
         DC.B   CCBEXTRP          This parameter specifies the number of
*                                 additional contiguous pages of memory
*                                 (256 bytes/page) to be assigned to each CCB
*                                 when each channel connected to the driver
*                                 is allocated by the CMR handler. The number
*                                 of pages allocated to the CCB is always the
*                                 value of this parameter plus one, so a
*                                 minimum of one page is always allocated.
         SPC    2
         DC.B   0,0,0             An additional 15 bytes are reserved for
         DC.L   0,0,0             future use by the CMR handler.
         SPC    2

         DC.L                     Optional revision information.
```

4.3.2 The Initialization Routine

The initialization routine is the second of the four elements that make up a standard device driver. This is the code section of the driver that is entered by the CMR ALLOCATE routine (in VERSAdos, the IOI task calls the CMR handler to allocate all channels). After a channel is established by the CMR handler, the I/O handler (the driver) is given a chance to do some initialization processing before normal execution resumes.

The calling sequence:

```
    ENTRY:    (SS) = Return PC into CMR handler
              A0   = Entry point for this routine
              A5   = CCB base address
    EXIT:     Must exit with RTS instruction
```

Only application-independent code must be executed in this routine (e.g., an initial device reset or a setting up of driver tables). If an I/O device is required only in a very specific way, the device setup or programming information could be coded and executed here. However, functions accomplished by this routine should not restrict the use of the driver. The initialization routine usually performs little action.

4.3.3 The Command Service Routine

This section of the driver performs most of the driver's work. The command service routine consists of a list of commands supported by the driver for each command. It is here that the actual I/O is initiated. The command service routine handles the INITIATE I/O, HALT, and RESET commands to start or terminate the necessary physical I/O.

The calling sequence:

ENTRY: (SS) = Return PC into the CMR routine
4 (SS) = Requester's TCB
8 (SS) = Status register for exit routine
10 (SS) = PC of exit routine
14 (SS) = Exit code
16 (SS) = Requester's status register
18 (SS) = Requester's PC
A0 = Self A2 = Physical address of requester's Parameter Block, either INITIATE I/O or HALT or RESET.
A5 = CCB Address
A6 = Requester's TCB address

EXIT: Must return with RTS instruction
D0 = Status value
A6 = Requester's TCB address

The INITIATE I/O command contains a pointer to the user's I/O parameter block. It is this parameter block that contains the commands specific to the driver, such as Read, Write, Configure, and Format.

For VERSAdos drivers these parameter blocks are described in the Data Management Services and Program Loader User's Manual.

For RMS68K drivers, the parameter block structure is unique for each driver.

4.3.4 The Interrupt Service Routine

This is the final element of the driver. It is at this routine that entry is accomplished in order to try to handle an interrupt (refer to the paragraph "Interrupts and Their Handlers" in this chapter). The first job of this routine is to establish whether the current interrupt came from the device whose driver this is. If ownership is not established, the routine returns to the CMR handler. However, if ownership is established, this section of the driver must handle or service the interrupt. Ownership of the interrupt is flagged to the CMR handler by writing the carry bit as follows:

carry = 1, interrupt serviced by this driver
carry = 0, this driver does not own this interrupt, try another driver

Because an I/O driver can accept interrupts directly from the device it controls, the EXEC relies on the handler to follow strict programming protocol so that system performance is not degraded. It is suggested that I/O handlers run at device priority only long enough to handle the cause of the interrupt, then lower the priority level by one and continue processing.

The calling sequence is as follows:

ENTRY: (SS) = PC of CMR return
4 (SS) = CCB address
8 (SS) = Register save A0-A2/D0
24 (SS) = PC of location after JSR instruction in CCB
28 (SS) = Status register at interrupt line
30 (SS) = PC at interrupt time
A0 = Work register
A1 = Self address
A5 = CCB address
D0 = Work register

EXIT: Must exit with RTS instruction
Must set or clear carry bit
All registers except A0, A1, A5, and D0 must be preserved

All I/O handlers will exit with an RTS instruction. The RTS will return to the common interrupt handler for possible preemptive processing. Figure 4-2 illustrates the driver structure.

VECTOR TABLE
AND
REVISION INFORMATION

INITIALIZATION
(CALLED WHEN CHANNEL IS ALLOCATED)

COMMAND SERVICE
(CALLED WHEN RESET, HALT I/O, OR INITIATE
I/O PARAMETER BLOCK RECEIVED)

INTERRUPT SERVICE
(CALLED WHEN AN INTERRUPT OCCURS ON THE INTERRUPT VECTOR)

Figure 4-2. Structure of a Driver

4.4 EQUATE FILES: IOE, TR1, TERMCCB, CCB, TCB

Several equate files are included with VERSAdos and RMS68K to provide a naming standard for the various data fields. The names and a description of these files are given below. These files are usually included but not listed at the head of utilities, applications software, and drivers.

. IOE	- This file contains all the required equates and a variety of macros for the I/O system, the TRAP #2 handler Input/Output Services (IOS), and the TRAP #3 handler File Handling Services (FHS).
. TR1	- This file contains the equates for the EXEC TRAP #1 calls. A TRAP #1 call macro is also included.
. TERMCCB	- This file contains all the device-dependent fields that relate to all serial port drivers. These fields are used by TERMLIB and by any serial port device driver that uses TERMLIB. The label marking the end of the generic terminal-dependent fields is TERMDDP.
. CCB	- This file contains equates for the device-independent part of the CCB. Other descriptive equates about channels are also given.
. TCB	- The Task Control Block equates can be found here.

These files can be included at the head of programs; their order is not important.

4.5 OBTAINING EXTRA MEMORY FOR THE DRIVER

The driver software may reside in system RAM or ROM. This choice of environment affects the way a driver can obtain memory for itself. The simplest and the preferred method is to request additional pages of CCB. These additional pages can then be used to maintain tables required by the driver and as scratchpad memory for the driver. Not all of the default page CCB may be defined, which may provide sufficient driver memory without additional pages. Using the CCB memory method ensures that the driver can be programmed into ROM. A second method to provide memory is to use the PAGEALOC TRAP #0 directive from the driver. This allows the driver to use a named memory segment as memory for itself (refer to Appendix B). This method also enables the driver to be programmed into ROM. A third memory acquisition method simply involves the driver declaring its own memory requirements within itself. However, this implies that the driver is located in a writable memory area, which may not always be desirable.

4.6 DRIVER CALLS TO RMS68K (TRAP #0)

Because drivers are processes, they run in supervisor mode of the MC68000. As such, TRAP #1 calls are not allowed. Therefore, to obtain various operating system services, a driver must use the TRAP #0 EXEC calls. The services made available to a process through TRAP #0 calls are listed below, and a full description of each function is provided in Appendix B.

EXRQPA

An Executive procedure may be scheduled or descheduled for activation on a periodic basis.

EXQEVENT

An I/O event may be queued to its associated task. The READY function, described below, is implicitly included.

PAUSE

While waiting for the completion of an I/O operation, an Executive procedure may relinquish its allocated time.

READY

An Executive procedure may change the state of a task from DORMANT to READY, in preparation for reactivation in accordance with the task's elected notification method. The READY function is implicit in both EXQEVENT and in WAKEUP.

WAKEUP

A task may be reactivated upon completion of a requested service. The READY function, described above, is implicitly included.

FNDGSEG

The address of a shared local or global memory segment may be retrieved from the Global Segment Table (GST) maintained by the operating system.

FNDTSEG

The address of a user or system task segment may be retrieved from the Task Segment Table (TST) maintained for each running task.

GETTCB

The address of a Task Control Block may be retrieved from the TCB table maintained by the operating system.

LOGPHY

The logical address of a memory segment may be converted to its physical address.

PAGEALOC

One or more pages of physical memory may be allocated to a task.

PAGEFREE

Page(s) of physical memory may be deallocated.

FNDUSEM

A task-associated semaphore may be located on a matching name basis or by association with the specified task.

PVSEM

A request for exclusive use of a resource may be queued or dequeued.

RDTIMER

The time of day may be read and returned.

KILLER

The cause of system trouble may be saved and the system deliberately crashed.

RMS68K TRAP #0 Call Summary

In the general course of events, a driver would use only a few of these calls, such as the queue-event call EXQEVENT, the logical-to-physical-conversion call LOGPHY, the allocate-memory-page call PAGEALOC, and the get-task-control-block call GETTCB.

The EXQEVENT queue-event call is used by the driver to queue a completion event to either IOS or the user task, depending on who called the driver. This is the standard exit from the driver's command service routine on completion of an I/O request.

The logical-to-physical-conversion call LOGPHY is used by the driver to convert a supplied logical address to an absolute physical address.

The allocate-physical-page(s)-of-memory call PAGEALOC is used to allocate one or more pages of physical memory to the task.

The get-task-control-block call GETTCB is used to obtain the address of a task's TCB from the TCB table.

CHAPTER 5

I/O WITH RMS68K

5.1 INTRODUCTION

This chapter discusses the mechanisms of I/O requests under RMS68K and lists a procedure for the steps required to SYSGEN a new driver into RMS68K.

5.2 EXECUTION OF AN I/O REQUEST WITHOUT USING IOS

The main difference between I/O requests with Input/Output Services (IOS) and those without is in the absence of the Data Control Block (DCB) in RMS68K situations. In addition, drivers running under RMS68K are faster than their counterparts running under VERSAdos, due to the absence of IOS. However, the user has a lot more to do when using RMS68K drivers and must be more careful. At this point, it must be stressed that drivers written to run under RMS68K will also run under VERSAdos, but the converse is not true.

The first step the user must take in a request for I/O is to prepare a CMR Initiate I/O Parameter Block (IIPB) which points to the correct data buffer (see Figure 5-1(a)). A TRAP #1 call is then issued with D0=60 to invoke the CMR handler. The CMR handler identifies the required Channel Control Block (CCB) and does a JSR into the driver (see Figure 5-1(b)). The driver inherits the pointers to IIPB and the CCB, and performs the requested I/O function (see Figure 5-1(c)). After completing the I/O request, the driver either queues a completion event to, or wakes up, the user task (see Figure 5-1(d)).

Because the DCB structure is nonexistent for a basic RMS68K system, this information, if required, can be given to the driver in the device-dependent portion of the CCB.

When RMS68K driver command service routines encounter any errors that prevent initiation of the required I/O, the driver should put an error code in D0 before returning to the CMR handler. If the associated parameter block contains a status field, the same error code should be put there also. The CMR handler will set the Z bit condition code of the status register to reflect the value of D0: 0 indicates acceptance of the parameter block and that I/O was started; nonzero indicates an error and that no I/O was started. This convention allows users to do a Branch-Equal to a good return or a Branch-Not-Equal for an error return. This is used with all RMS68K TRAP #1 calls.

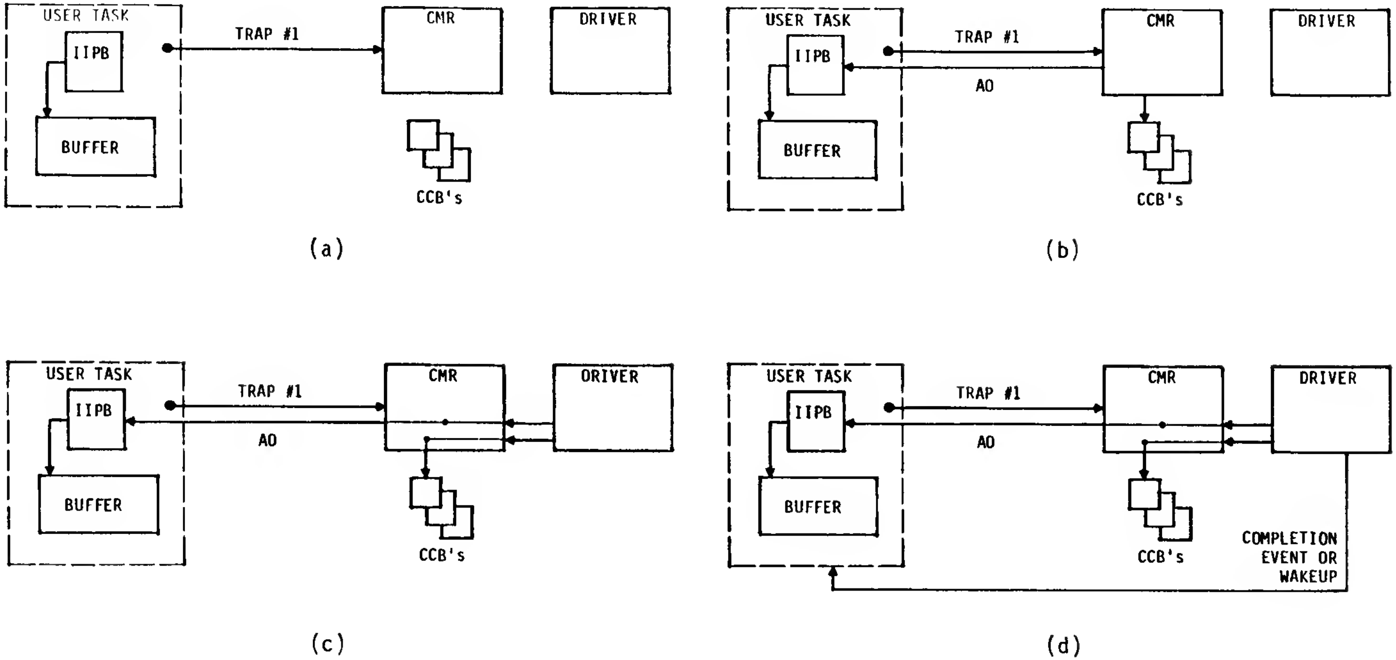

USER TASK
IIPB
BUFFER
TRAP #1
CMR
DRIVER
CCB's
(a)
USER TASK
IIPB
BUFFER
TRAP #1
A0
CMR
DRIVER
CCB's
(b)
USER TASK
IIPB
BUFFER
TRAP #1
A0
CMR
DRIVER
CCB's
(c)
USER TASK
IIPB
BUFFER
TRAP #1
A0
CMR
DRIVER
CCB's
COMPLETION
EVENT OR
WAKEUP
(d)

FIGURE 5-1. Executing an I/O Request without Using IOS

5.3 A DRIVER ADDITION ALGORITHM FOR SYSGENING RMS68K DRIVERS

An RMS68K driver may also be referred to as a process control driver.

Adding a driver to RMS68K requires three main steps: adding the driver code itself; adding the Channel Data Block (CDB) information, together with the I/O Channel initializer task; and adding code to the system initializer process if the driver requires memory for itself in the SYSPAR region of low memory.

The following paragraphs describe a step-by-step procedure for adding a new RMS68K driver to VERSAdos.

During this process, several new files must be created, as described in the list below. In this list, xxxxDRV is the name of the driver, where xxxx designates the module number for Motorola module products. For example, the file 9992.&.M610DRV.SA is the source file for the MVME610/620 analog-to-digital converter module.

NOTE

Conventions have been established for the use of the filename extensions .SA, .AG, .CI, .LG, .AF, and .SI, used by the SYSGEN files. Refer to the System Generation Facility User's Manual for details.

The following files must be created. Appendix F contains examples of these files.

FILENAME	DESCRIPTION
9992.&.xxxxDRV.SA	Source file for the driver.
9992.&.xxxxDRV.AF	Assembly chainfile for the driver. This file should include comments listing any INCLUDE files called in by the driver.
9992.&.xxxxDRV.RO	Relocatable object file for the driver. Created by the command =CHAIN &.xxxxDRV.AF.
9992.&.xxxxDRV.LG	Link file for the driver.
9992.&.xxxxIOC.SI	File which contains CDBs for the driver.
9998.VERSAPT.xxxxDRV.CF	Patch file for the driver.
9992.&.xxxxDRV.CI	INCLUDE file for SYSGENing in the driver.
9992.&.xxxxMEM.CI	INCLUDE file for bringing in the memory allocation module.
9992.&.xxxxMEM.AG	Memory allocation module (contains code to allocate memory).
9998.COPYGEN.xxxxDRV.CF	Chainfile for copying driver files into a user number where SYSGEN is to be performed.

Also, several existing files must be modified. In the following list, which describes them, <system> refers to the catalog name for the target system. For example, the MVME110 target system files are under the VME110 catalog name, and the VME/10 target system files are under the VMES10 catalog name.

The following files must be modified:

FILENAME	DESCRIPTION
9998.<system>.CNFGDRVR.CI	Switch file of modules in the system.
9998.<system>.IFDRVR.CI	Conditional file to bring in &.xxxxDRV.CI.
9992.&.PCDRV.CI	Conditional file based on switch values in <system>.CNFGDRVR.CI to bring in the file &.xxxxMEM.CI.
9998.<system>.COPYSGEN.CF	Chainfile to copy all files for a system SYSGEN into a particular user number.
9998.IOC.ADDRESS.CI	I/O Channel address offsets (base address depends on system).
9992.&.IOCINT.AG	File which contains CDBs.

The procedure for loading a new driver into the operating system is listed below. Refer to Appendix F for examples of the files created or modified during each step.

a. Create files &.xxxxDRV.SA, &.xxxxDRV.AF, and &.xxxxDRV.RO.

   Write the I/O driver code so that it conforms to the 4-element standard driver structure. Using a similar type of RMS68K driver as a model, create the source file, assembly chainfile, and relocatable object file.

b. Create the file &.xxxxDRV.LG.

   This is the file that links the driver into the system.

   The following SYSGEN commands:

```
SUBS   &.xxxxDRV.LG
LINK   &.xxxxDRV.LG
```

   are found in the file &.xxxxDRV.CI.

The link file for SYSGEN should have a command line such as the following for the linker:

```
=LINK ,&.xxxxDRV.LO,\LINKLS;<options>
SEG xxxx:nn  \xxxxDRV
INPUT &.xxxxDRV.RO
INPUT <any other files needed>
END
=END
```

The MVME600 modules, for example, must be linked with &.SYSPAR.RO.

For a guide to comments and options, refer to the driver model or to Appendix F.

c. Create the file &.xxxxIOC.SI.

In the file &.xxxxDRV.CI is the SYSGEN command

```
SUBS   &.xxxxIOC.SI
```

which will create a new file called &.XxxxxIOC.SI. This new file will substitute the defined SYSGEN parameters with their values wherever they occur in xxxxIOC.SI.

The file &.XxxxxIOC.SI is included in the file &.IOCINT.AG if the switch for the driver is set in the <system>.CNFGDRVR.CI file.

&.xxxxIOC.SI contains the CDB setup information for the module. The CDB macro (refer to Appendix D) should be used at this point in the procedure.

d. Create the file VERSAPT.xxxxDRV.CF.

This is the patch file for the driver.

e. Create the file &.xxxxDRV.CI.

This is the file that will bring the driver and its device configuration into the system. A conditional assembly in <system>.IFDRVR.CI will bring it in if the appropriate switches in <system>.CNFGDRVR.CI have been set.

&.xxxxDRV.CI performs the SUBS command on the file xxxxIOC.SI, and the SUBS and LINK commands on the file xxxxDRV.LG. It also appends the file VERSAPT.xxxxDRV.CF to the file <system>.VERSAPT.CF.

f. Create the files &.xxxxMEM.CI and &.xxxxMEM.AG.

Most process control drivers require some memory allocation at system initialization time. If one of the process control module switches has been set, then the PCDRV switch will be set (in xxxxDRV.CI), and the file &.PCDRV.CI will be pulled into the system within the file &.SYSINIT.CI. For each process control module whose switch is set, the file &.xxxxMEM.CI will be included. Within the file &.xxxxMEM.CI are commands to assemble the file &.xxxxMEM.AG.

As each .RO module is created, it is linked into a temporary .RO file. When all .RO files are linked together, a final link creates the file <system>.SYSINIT.LO.

One of the process control drivers should be used as a model.

g. Create the file COPYGEN.xxxxDRV.CF.

This file will be included in <system>.COPYSGEN.CF for each system that can use the driver. It allows easy access to all the SYSGEN files pertaining to the driver.

h. Modify the file <system>.CNFGDRVR.CI.

This is a switch file that allows users to select the modules to be included in the system.

i. Modify the file <system>.IFDRVR.CI.

This is a conditional assembly file, based on the switches set in <system>.CNFGDRVR.CI, to include &.xxxxDRV.CI if the module is to be used in the system.

j. Modify IOC.ADDRESS.CI.

Choose the address of a readable register on the module, because this is the address that is read during boot to determine if the module is there. Also, this is the address that the driver looks for at the CCBCHB offset in the CCB when accessing the device.

k. Modify &.IOCINT.AG.

This file contains the CDBs for the process control drivers. In &.IOCINT.AG there are conditional assemblies, based on the switch settings in file <system>.CNFGDRVR.CI, to bring in the file &.XxxxxIOC.SI.

l. Modify the file &.PCDRV.CI.

   This file contains the &.xxxxMEM.CI files for memory allocation. The &.PCDRV.CI file has conditional assemblies based on the status of switches in the <system>.CNFGDRVR.CI file. These conditional assemblies cause the appropriate &.xxxxMEM.CI file to be included for each process control module for which a switch is set (refer to step f.).

m. Modify <system>.COPYSGEN.CF.

   This is the chainfile that copies to a designated user area all the files needed for a SYSGEN for the target system. This chainfile includes the COPYGEN.xxxxDRV.CF file if the system is to use the module.

5.4 MAP OF INCLUDE FILES IN SYSGEN FILES (WITH PROCESS CONTROL DRIVER)

The following is a list of INCLUDE files used during the process of SYSGENing a new driver into RMS68K.

&.VERSADOS.CD:

```
    INCLUDE     <system>.RMS.CI
    INCLUDE     &.DRVLIB.CI
    INCLUDE     &.TERMLIB.CI
    INCLUDE     <system>.SYSTEM.CI
    INCLUDE     &.CNFGTASK.CI
    INCLUDE     &.VALPAR.CI
    INCLUDE     <system>.CNFGDRVR.CI
    INCLUDE     <system>.IFDRVR.CI
    INCLUDE     &.IOCGEN.CI
    INCLUDE     &.IFTASK.CI
    INCLUDE     &.IOI.CI
    INCLUDE     &.IOCI.CI
    INCLUDE     &.SYSINIT.CI
```

<system>.SYSTEM.CI:

```
    INCLUDE     SIO.ADDRESS.CI
    INCLUDE     IOC.ADDRESS.CI
```

<system>.IFDRVR.CI:

```
    INCLUDE     &.xxxxDRV.CI
```

&.IFTASK.CI:

```
    INCLUDE     FHSIOS.VERSADOS.CI
    INCLUDE     FMS.VERSADOS.CI
    INCLUDE     EET.VERSADOS.CI
    INCLUDE     MMULDR.VERSADOS.CI            (for MMU systems)
    INCLUDE     NOMMULDR.VERSADOS.CI          (for non-MMU systems)
```

```
&.xxxxDRV.CI:
    &PCDRV   =  \&PCDRV+1
    SUBS        &.xxxxIOC.SI
    SUBS        &.xxxxDRV.LG
    LINK        &.xxxxDRV.LG
    PROCESS     &.xxxxDRV.LO
    END         &.xxxxDRV.LO

& .IOCI.CI:
    TASK        &.IOCI.LO
    SUBS        &.IOCINT.AG
    ASM         &.IOCINT.AG,&.IOCINT.RO,\ASMLS;Z=100
    SUBS        &.IOCINT.LG
    LINK        &.IOCINT.LG
    END         &.IOCI.LO

&.IOCINT.AG:
    INCLUDE     XxxxxIOC.SI

&.SYSINIT.CI:
    INCLUDE     &.PCDRV.CI

&.PCDRV.CI:
    INCLUDE     &.xxxxMEM.CI

&.xxxxMEM.CI:
    SUBS        &.xxxxMEM.AG
    ASM         &.xxxxMEM.AG,NEW.RO,\ASMLS
    INCLUDE     &.ROGEN.CI

&.ROGEN.CI:
    LINK        &.ROGEN.LG
```

CHAPTER 6

I/O WITH VERSAdos

6.1 INTRODUCTION

This chapter describes the mechanisms of I/O requests under VERSAdos and lists the steps required to SYSGEN a new driver into VERSAdos.

6.2 EXECUTION OF AN I/O REQUEST USING IOS

What follows is a description, in both words and pictures, of I/O execution using IOS. This procedure is applicable only in a VERSAdos environment. Refer to Figure 6-1 during the following discussions.

After deciding to perform I/O, the user must first set up an IOS I/O block (IOCB). The IOCB points to the user's buffer. This buffer will contain the data to be written or will be where the data will be put for a read. The user task then issues a TRAP #2 to IOS (see Figure 6-1(a)). IOS then takes control for the second step of the process. IOS prepares an Initiate I/O Parameter Block (IIPB) which will point back to the user's task IOCB and also to the Device Control Block (DCB) for the required I/O device. IOS then does a TRAP #1 call to the CMR handler (with D0=60); see Figure 6-1(b). The CMR handler identifies the Channel Control Block (CCB) and, using a JSR instruction, moves control to the device driver (see Figure 6-1(c)). The driver inherits the pointers to IIPB and the CCB, and performs the requested I/O function in its command service routine, shown in Figure 6-1(d). After completing the I/O request, the driver updates the IOCB and queues a completion event to IOS (see Figure 6-1(e)). IOS then performs an acknowledge request (AKRQST) to the calling user task or queues an event to it, depending on the WAIT/PROCEED mode selected by the user in the IOCB (see Figure 6-1(f)).

Parameter block error reporting in VERSAdos drivers is handled by queuing an event to the requester of the I/O.

6.3 A DRIVER ADDITION ALGORITHM FOR SYSGENING VERSAdos DRIVERS

This paragraph relates to I/O drivers that are entered using IOS (TRAP #2) under VERSAdos. A step-by-step procedure for adding a new driver to VERSAdos follows. Figure 6-2 provides an overview of this procedure.

During this procedure several new files must be created, as described in the list which follows. In this list, xxxxDRV is the name of the driver, where xxxx designates the module number for Motorola module products, whenever applicable. For example, the file 9993.&.M435DRV.SA is the source file for the magnetic tape driver on the MVME435 magnetic tape adapter module.

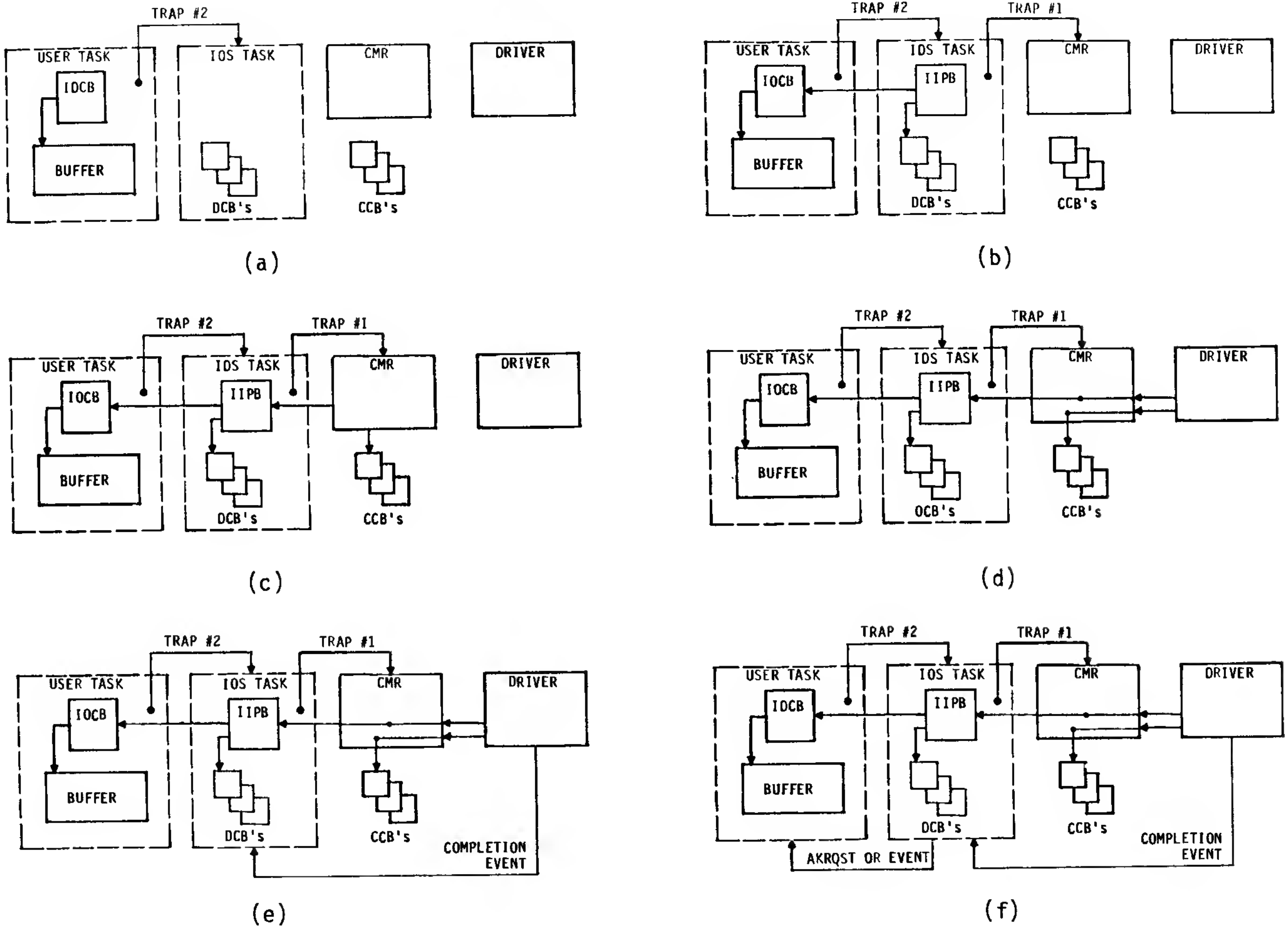
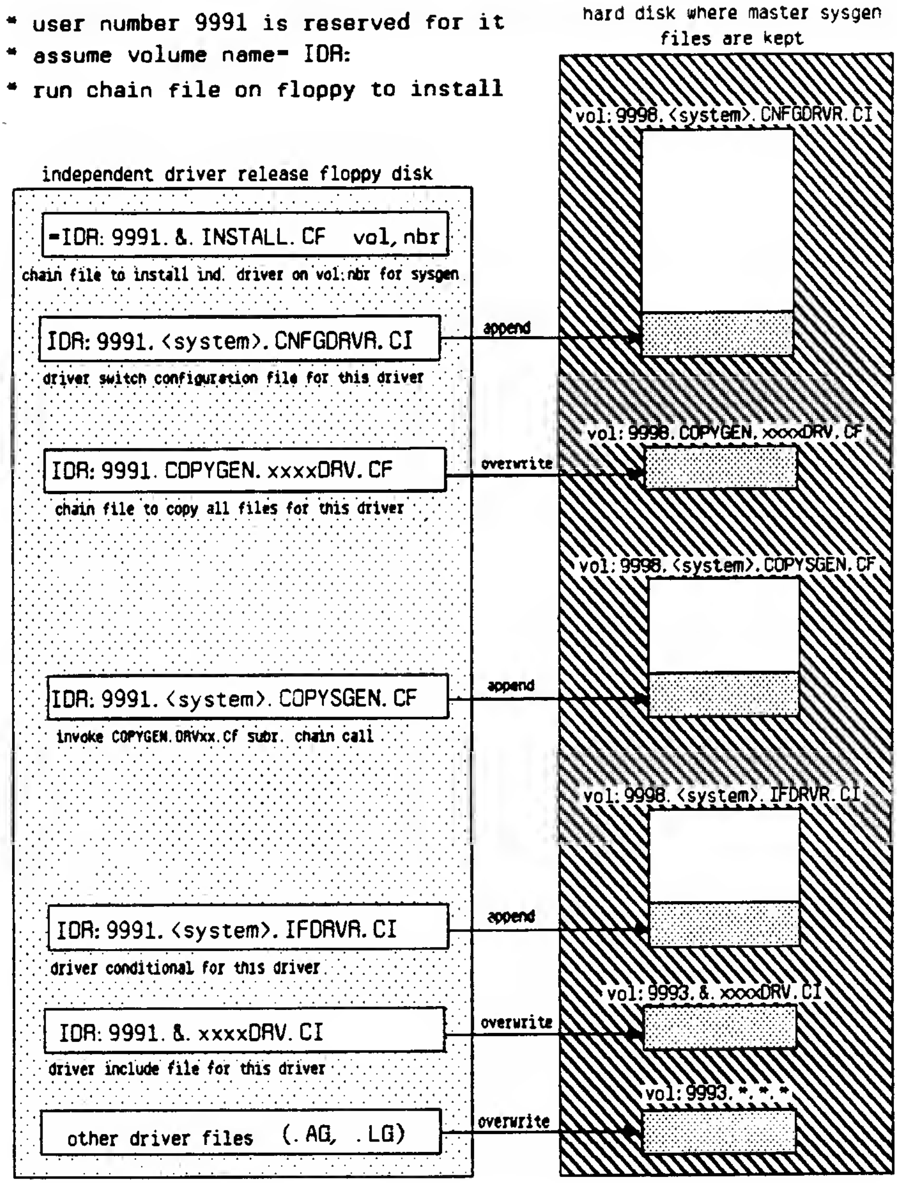

TRAP #2
USER TASK
IOCB
BUFFER
IOS TASK
DCB's
CMR
CCB's
DRIVER
(a)
TRAP #2
TRAP #1
USER TASK
IOCB
BUFFER
IOS TASK
IIPB
DCB's
CMR
CCB's
DRIVER
(b)
TRAP #2
TRAP #1
USER TASK
IOCB
BUFFER
IOS TASK
IIPB
DCB's
CMR
CCB's
DRIVER
(c)
TRAP #2
TRAP #1
USER TASK
IOCB
BUFFER
IOS TASK
IIPB
OCB's
CMR
CCB's
DRIVER
(d)
TRAP #2
TRAP #1
USER TASK
IOCB
BUFFER
IOS TASK
IIPB
DCB's
CMR
CCB's
DRIVER
COMPLETION EVENT
(e)
TRAP #2
TRAP #1
USER TASK
IOCB
BUFFER
IOS TASK
IIPB
DCB's
CMR
CCB's
DRIVER
AKRQST OR EVENT
COMPLETION EVENT
(f)

FIGURE 6-1. Executing an I/O Request Using IOS

INSTALLATION OF INDEPENDENT DRIVER RELEASE

Microsystems 08 Mar. '85

- on 5-1/4" or 8" floppy only
- user number 9991 is reserved for it
- assume volume name= IDR:
- run chain file on floppy to install



hard disk where master sysgen files are kept

independent driver release floppy disk

=IDR: 9991. &. INSTALL. CF vol, nbr

chain file to install ind. driver on vol:nbr for sysgen

IDR: 9991. <system>. CNFGDRVR. CI

driver switch configuration file for this driver

IDR: 9991. COPYGEN. xxxxDRV. CF

chain file to copy all files for this driver

IDR: 9991. <system>. COPYSGEN. CF

invoke COPYGEN.DRVxx.CF subr. chain call

IDR: 9991. <system>. IFDRVR. CI

driver conditional for this driver

IDR: 9991. &. xxxxDRV. CI

driver include file for this driver

other driver files (. AG, . LG)

append
overwrite
append
append
overwrite
overwrite
vol: 9998. <system>. CNFGDRVR. CI
vol: 9998. COPYGEN. xxxxDRV. CF
vol: 9998. <system>. COPYSGEN. CF
vol: 9998. <system>. IFDRVR. CI
vol: 9993. &. xxxxDRV. CI
vol: 9993. *. *. *

FIGURE 6-2. Overview of Driver Installation into VERSAdos

NOTE

> Conventions have been established for the use of the filename extensions .SA, .AG, .CI, .LG, .AF, and .SI, used by the SYSGEN files. Refer to the System Generation Facility User's Manual for details.

The following files must be created. Appendix F contains examples of these files.

FILENAME	DESCRIPTION
9993.&.xxxxDRV.SA	Source file for the driver.
9993.&.xxxxDRV.AF	Assembly chainfile for the driver. This file should include comments listing any INCLUDE files called by the driver.
9993.&.xxxxDRV.RO	Relocatable object file for the driver. Created by the following command: =CHAIN &.xxxxDRV.AF.
9993.&.xxxxDRV.LG	Link file for the driver.
9998.IOC.xxxxDRV.AG	Sets up the DCBs and CDBs for the driver.
9998.VERSAPT.xxxxDRV.CF	Patch file for the driver.
9993.&.xxxxDRV.CI	INCLUDE file for SYSGENing in the driver.
9998.COPYGEN.xxxxDRV.CF	Chainfile for copying driver files into a user number where SYSGEN is to be performed.

Also, several existing files must be modified. In the following list, which describes them, <system> refers to the catalog name for the target system. For example, the MVME110 target system files are under the VME110 catalog name, and the VME/10 target system files are under the VMES10 catalog name.

The following files must be modified:

FILENAME	DESCRIPTION
9998.<system>.CNFGDRVR.CI	Switch file of modules in the system.
9998.<system>.IFDRVR.CI	Conditional file based on switch values in <system>.CNFGDRVR.CI.
9998.<system>.COPYSGEN.CF	Chainfile to copy all files for a system SYSGEN into a particular user number.

One of the following files must be chosen according to whether or not the module is an I/O Channel module:

9998.IOC.ADDRESS.CI	I/O Channel address offsets (base address depends on system).
or	
9998.SIO.ADDRESS.CI	Short I/O space address offsets.

The procedure for loading a new driver into the operating system follows. Refer to Appendix F for examples of the files created during each step.

a. Create files &.xxxxDRV.SA, &.xxxxDRV.AF, and &.xxxxDRV.RO.

   Write the I/O driver code to conform to the 4-element standard driver structure. Using a similar VERSAdos driver as a model, create the source file, assembly chainfile, and relocatable object file.

   When writing a serial port driver using TERMLIB, refer to Chapter 7.

b. Create the file &.xxxxDRV.LG.

   The link file for SYSGEN should have a command line for the linker like the following:

```
=LINK ,&.xxxxDRV.LO,\LINKLS;<options>
SEG xxxx:0 \xxxxDRV
INPUT &.xxxxDRV.RO
INPUT <any other files needed>
END
=END
```

   For a guide to comments and options, refer to the driver model.

   When writing a serial port driver using TERMLIB, refer to Chapter 7.

c. Create the file IOC.xxxxDRV.AG.

   This file sets up the DCBs and CDBs for the module. In the file &.xxxxDRV.CI are the SYSGEN commands

```
SUBS      IOC.xxxxDRV.AG
ASM       IOC.xxxxDRV.AG,NEW.RO,\ASMLS;RMZ=85
```

   The SUBS command creates a file named IOC.XxxxxDRV.AG, which replaces the defined SYSGEN parameters with their values wherever they occur in IOC.xxxxDRV.AG.

   The ASM command assembles the file IOC.XxxxxDRV.AG. This file looks different for different devices.

   Refer to Appendices D and E, and to the System Generation Facility User's Manual, for a detailed description of the macros used to set up the DCBs and CDBs.

DISK DRIVERS

When writing a disk driver, a series of SET directives is used to set up the fields in the DSKDCB macro, as follows:

```
*
*  Set up DCB parameters for RWIN1 (media independent)
*
DEV_ATT  SET       $1F       Supports read, write, binary, random, image.
DEV_CODE SET       0         Has no meaning for disk drivers.
DEV_STAT SET       4         "Device status changed" works for disks.
CHAN_ID  SET       'WIND'    Channel mnemonic.
PAR_MASK SET       $1AF3     Mask sent down for first Configure call.
*                            Refer to the Data Management Services Manual
*                            and set up the parameters mask according
*                            to the parameters supported  by the driver.
*                            Note that the read time-outs and write
*                            time-outs are 0 (bits 2 and 3) because
*                            these values cannot be changed in a
*                            Configure call.
ECC_LEN SET       0          ECC data burst length parameter.
```

The attributes word is media dependent, and is defined in the media file. However, the alternate sectors bit in the attributes word is also controller dependent, so that bit is set after both the type of disk and the type of controller is known.

```
*
*  Set up attribute mask for hard disk on RWIN1.  These
*  are the attributes that are recognized by the RWIN
*  controller and that are legal for a Configure call.
*

ATT_MASK SET  0<<IOADDEN+0<<IOATDEN+0<<IOADSIDE+0<<IOAFRMT
ATT_MASK SET  ATT_MASK+1<<IOARDISC+0<<IOADDEND+0<<IOATDEND+0<<IOARIBS
ATT_MASK SET  ATT_MASK+0<<IOADPCOM+0<<IOASIZE+1<<IOAALT

ATT_WORD SET  ATT_WORD+1<<IOAALT
*
*  Set up interleave for hard disk on RWIN1.
*
INTERLEAVE SET     1
```

It must be decided which types of disks will be supported for each drive number. For example, the RWIN driver supports the following:

Hard disks (Winchesters):

Drive 0: (#HDx0)

8" 10Mb hard disk
5-1/4" 5Mb hard disk
5-1/4" 10Mb hard disk
5-1/4" 15Mb hard disk
5-1/4" 40Mb hard disk

Drive 1: (#HDx1)
8" 10Mb hard disk
5-1/4" 5Mb hard disk
5-1/4" 10Mb hard disk
5-1/4" 15Mb hard disk
5-1/4" 40Mb hard disk

Floppies:
Drive 2: (#FDx2)
8" double-density, double-sided, IBM format
8" single-density, double-sided, Motorola format
8" single-density, single-sided, IBM format
8" single-density, single-sided, Motorola format
5-1/4" double-density, double-sided, IBM format

Drive 3: (#FDx3)
8" double-density, double-sided, IBM format
8" single-density, double-sided, Motorola format
8" single-density, single-sided, IBM format
8" single-density, single-sided, Motorola format
5-1/4" double-density, double-sided, IBM format

Conditional assembly is performed with flags that are set in <system>.CNFGDRVR.CI to set up each drive number with the user-selected type of disk.

For example, if the user wants the following disk configuration:

> System has one RWIN controller module. RWIN is controller 0.
> There are two hard disk drives and two floppy disk drives.
> #HD00 is a 15Mb 5-1/4" Winchester,
> #HD01 is a 40Mb 5-1/4" Winchester,
> #FD02 is a dbl-density, dbl-sided 5-1/4" IBM format floppy,
> #FD03 is a dbl-density, dbl-sided 5-1/4" IBM format floppy

Then in <system>.CNFGDRVR.CI the following code must appear:

```
           NORWIN = 1         of RWIN1 Winchester controller modules
           IFGT   \NORWIN
                     CONTWIN1  =  "0"    RWIN1 is controller 0
                     NHRWIN$1  =   2     2 hard disk drives
                     NFRWIN$1  =   2     2 floppy disk drives

                     RWIN0$1  =  "'H5WIN15'" first RWIN, drive 0
*                                 is 15Mb 5-1/4"
                     RWIN1$1  =  "'H5WIN40'" first RWIN, drive 1
*                                 is 40Mb 5-1/4"
                     RWIN2$1  =  "'F5DDDSI'" first RWIN, drive 2
*                                 is floppy, 5-1/4", double-data-
*                                 density, double-sided, IBM format
                     RWIN3$1  =  "'F5DDDSI'" first RWIN, drive 3
*                                 is floppy, 5-1/4", double-data-
*                                 density, double-sided, IBM format
           ENDC
```

In IOC.RWINDRV.AG, there are conditional assemblies to include the proper media configuration file for the type of media selected in <system>.CNFGDRVR.CI.

For example, for the first hard disk on the first RWIN controller the code is:

```
     IFC    \RWIN0$1,'H5WIN15'
            INCLUDE   &.H5WIN15.SI
     ENDC
```

The file &.H5WIN15.SI sets fields in the DCB that are media dependent and reflect the attributes and parameters for a 5-1/4 inch, 15Mb Winchester hard disk.

The following symbols are used to define the media selected for a disk drive. (Win indicates Winchester type.)

SYMBOL	FILENAME	DISK DESCRIPTION
'H8WIN10'	&.H8WIN10.SI	Hard, Win, 8", 10Mb
'H5WIN05'	&.H5WIN05.SI	Hard, Win, 5-1/4", 5Mb
'H5WIN10'	&.H5WIN10.SI	Hard, Win, 5-1/4", 10Mb
'H5WIN15'	&.H5WIN15.SI	Hard, Win, 5-1/4", 15Mb
'H5WIN40'	&.H5WIN40.SI	Hard, Win, 5-1/4", 40Mb
'F8SDDSM'	&.F8SDDSM.SI	Floppy, 8", single-data-density, double-sided, Motorola format
'F8SDSSM'	&.F8SDSSM.SI	Floppy, 8", single-data-density, single-sided, Motorola format
'F5DDDSI'	&.F5DDDSI.SI	Floppy, 5-1/4", double-data-density, double-sided, IBM format
'RMCMD16'	&.RMCMD16.SI	Removable CMD 16Mb
'FXCMD16'	&.FXCMD16.SI	Fixed CMD 16Mb
'FXCMD80'	&.FXCMD80.SI	Fixed CMD 80Mb
'RMLRK08'	&.RMLRK08.SI	Removable LARK 8Mb
'FXLRK08'	&.FXLRK08.SI	Fixed LARK 8Mb
'RMLRK25'	&.RMLRK25.SI	Removable LARK 25Mb
'FXLRK25'	&.FXLRK25.SI	Fixed LARK 25Mb
'F8DDDSI'	&.F8DDDSI.SI	Floppy 8", double-data-density, double-sided, IBM format
'F8SDSSI'	&.F8SDSSI.SI	Floppy 8", single-data-density, single-sided, IBM format

TERMINAL AND PRINTER DRIVERS

Each type of driver has a slightly different format in the file IOC.xxxxDRV.AG. When writing a driver for a terminal or printer use a similar type of driver as a model and change it as needed to fit the specifications of the driver being written.

NOTE

> The DCBs will end up in the shared segment IOSG, whereas the CCBs exist along with the TCB structures in the EXEC system space. Using the utility SYSANAL, both the DCB and CCB structures can be examined in a "live" VERSAdos system.

d. Create the file VERSAPT.xxxxDRV.CF.

   This is the patch file for the driver.

e. Create the file &.xxxxDRV.CI.

   This is the file that brings the driver and its device configuration into the system. A conditional assembly in <system>.IFDRVR.CI brings it in if the appropriate switches in <system>.CNFGDRVR.CI are set.

   File &.xxxxDRV.CI uses the files &.xxxxDRV.LG, IOC.xxxxDRV.AG, and VERSAPT.xxxxDRV.CF.

f. Create the file COPYGEN.xxxxDRV.CF.

   This file is included in <system>.COPYSGEN.CF for each system that can use the driver.

g. Modify the file <system>.CNFGDRVR.CI.

   This is a switch file that allows users to select the modules to be included in the system.

h. Modify the file <system>.IFDRVR.CI.

   This is a conditional assembly file, based on the switches set in <system>.CNFGDRVR.CI, to include &.xxxxDRV.CI if the module is to be used in the system.

i. Modify either IOC.ADDRESS.CI or SIO.ADDRESS.CI.

   For an I/O Channel module, add the address of the module to IOC.ADDRESS.CI. Otherwise, add the address of the module to SIO.ADDRESS.CI. The address space for the module should not overlap with another module that will be included in the system.

   Choose an address of a readable register on the module, because this is the address that is read during boot to determine if the module is there. Also, this is the address that the driver finds at the CCBCHB offset in the CCB when accessing the device.

j. Modify <system>.COPYSGEN.CF.

   This is the chainfile that copies to a designated user area all the files needed for a SYSGEN for the target system. This chainfile includes file COPYGEN.xxxxDRV.CF if the system is to use the module.

6.4 MAP OF INCLUDE FILES IN THE SYSGEN FILES (WITHOUT PROCESS CONTROL DRIVER)

The following is a list of INCLUDE files used during the process of SYSGENing a new driver into VERSAdos.

```
&.VERSADOS.CD:

     INCLUDE   <system>.RMS.CI
     INCLUDE   &.DRVLIB.CI
     INCLUDE   &.TERMLIB.CI
     INCLUDE   <system>.SYSTEM.CI
     INCLUDE   &.CNFGTASK.CI
     INCLUDE   &.VALPAR.CI
     INCLUDE   <system>.CNFGDRVR.CI
     INCLUDE   <system>.IFDRVR.CI
     INCLUDE   &.IOCGEN.CI
     INCLUDE   &.IFTASK.CI
     INCLUDE   &.IOI.CI
     INCLUDE   &.SYSINIT.CI

<system>.SYSTEM.CI:
     INCLUDE   SIO.ADDRESS.CI
     INCLUDE   IOC.ADDRESS.CI

<system>.IFDRVR.CI:
     INCLUDE   &.xxxxDRV.CI

&.IFTASK.CI
     INCLUDE   FHSIOS.VERSADOS.CI
     INCLUDE   FMS.VERSADOS.CI
     INCLUDE   EET.VERSADOS.CI
     INCLUDE   MMULDR.VERSADOS.CI              (for MMU systems)
     INCLUDE   NOMMULDR.VERSADOS.CI            (for non-MMU systems)


&.xxxxDRV.CI:
     SUBS      &.xxxxDRV.LG
     LINK      &.xxxxDRV.LG
     PROCESS   &.xxxxDRV.LO
     END       &.xxxxDRV.LO
     SUBS      IOC.xxxxDRV.AG
     ASM       IOC.xxxxDRV.AG,NEW.RO,\ASMLS;RMZ=85
     INCLUDE   &.IOCGEN.CI
     LINK      &.IOCGEN.LG
```

CHAPTER 7

TERMLIB

7.1 INTRODUCTION

The file TERMLIB was created to ease the task of writing drivers under the VERSAdos operating system by defining device-independent routines required. This chapter describes the structure of TERMLIB and the routines available therein. It also explains how to write a driver using TERMLIB and how to incorporate the new driver into the operating system.

7.2 DRIVER ROUTINE FILES

Three files contain the code required by drivers under VERSAdos. These files, and their contents, are

DRVLIB	routines common to all drivers
TERMLIB	device-independent code and routines common to all serial port drivers
MPSCDRV	device-dependent code

The routines in DRVLIB and TERMLIB are described in the following paragraphs.

7.3 DRVLIB ROUTINES

The DRVLIB file contains the following routines, which are common to all drivers:

QEVENT
LOGPHY
SET_TIME

7.3.1 QEVENT

QEVENT is a collection of subroutines that queue events from the driver to a task (usually IOS). The types of events queued are

- Normal I/O completion event
- HALT I/O event
- Unsolicited device event where DCB address is known

There are six different subroutines in the QEVENT module, which allow the queuing of each event type listed from either the command level or the interrupt handler.

7.3.1.1 QEVENT Subroutine Descriptions.

SUBROUTINE	DESCRIPTION OF EVENT TO BE SENT
N_NRM_QEVENT	Normal I/O completion event Queued from command level (0)
I_NRM_QEVENT	Normal I/O completion event Queued from interrupt handler
N_HLT_QEVENT	HALT I/O event Queued from command level (0)
I_HLT_QEVENT	HALT I/O event Queued from interrupt handler
N_UNS_QEVENT	Unsolicited device event Queued from command level (0) Status value 0 (DCB known)
I_UNS_QEVENT	Unsolicited device event Queued from interrupt handler Status value 0 (DCB known)

7.3.1.2 Entry and Exit Conditions and Register Usage. These paragraphs describe entry and exit conditions and register usage for QEVENT subroutines.

Entry Points for Normal I/O Completion and HALT I/O Events

Subroutine names:

N_NRM_QEVENT
I_NRM_QEVENT
N_HLT_QEVENT
I_HLT_QEVENT

Entry:
A0 = address of place to copy event (must have at least 20 bytes reserved)
or
0 if the event will not be copied
A5 = address of CCB
D1.B = error message
D3.L = address of DCB

Registers usage:

	0	1	2	3	4	5	6	7
Data	S	P	S	P	S	S	S	S
Address	P	S	S	S	S	P		

P = parameter register
R = return register
* = destroyed register
S = saved and restored register

Exit: Condition Code Register (CCR):
<EQ> = TRAP #0 call succeeded
<NE> = TRAP #0 call did not succeed

Exits to: RTS to calling routine

Entry Points for Unsolicited Device Event (Status Value 0)

Subroutine names:

N_UNS_QEVENT
I_UNS_QEVENT

Entry: A0 = address of place to copy event (must have at least 20 bytes reserved)
or
0 if the event will not be copied
A5 = address of CCB
D3.L = address of DCB
D4.W = device status and type

Registers usage:

	0	1	2	3	4	5	6	7
Data	S	S	S	P	P	S	S	S
Address	P	S	S	S	S	P		

P = parameter register
R = return register
* = destroyed register
S = saved and restored register

Exit: CCR: <EQ> = TRAP #0 call succeeded
<NE> = TRAP #0 call did not succeed

Exits to: RTS to calling routine

7.3.1.3 QEVENT Examples. The following paragraphs contain examples of the use of QEVENT subroutines. Each example assumes a field in the CCB called DCB_ADDR that holds the address of the DCB.

Normal Event From Interrupt Handler

In this example, register A5 contains the address of the CCB.

```
CLR.L     A0                  Event not to be copied.
MOVE.B    #ISTAOK,D1          "OK" message (no error).
MOVE.L    DCB_ADDR(A5),D3     D3 <--- DCB address.
BSR.L     I_NRM_QEVENT        BSR to the routine.
IF  <NE>  THEN                If something went wrong,
   TRO$.KILLER ,              kill the system.
ENDI
```

Normal Event from Commands (Level 0)

In this example, register A5 contains the address of the CCB.

```
    CLR.L      A0                    Event not to be copied.
    MOVE.W     #ISTAIF,D1            Command not found error.
    MOVE.L     DCB_ADDR(A5),D3       D3 <--- DCB address.
    BSR.L      N_NRM_QEVENT          BSR to the routine.
    IF  <NE>   THEN                  If something went wrong,
       TRO$.KILLER ,                 kill the system.
    ENDI
```

Unsolicited Device Event from Interrupt Handler

In this example, register A5 contains the address of the CCB. Also, the subroutine GET_STATUS polls the device and returns its status in D0.B.

```
    CLR.L      A0                    Event not to be copied.
    BSR.L      GET_STATUS            D0.B <--- Device status.
    MOVE.B     D0,D4                 D4.B <---  Device status.
    LSL.W      #8,D4                 Shift into second byte.
    MOVE.B     #XDSACIA,D4           D4.B <---  Device type.
*
*  Now register D4 contains the following:
*                       +---------+---------+---------+---------+
*                       |    ?    |    ?    | dev stat |dev type |
*                       +---------+---------+---------+---------+
*
    MOVE.L     DCB_ADDR(A5),D3       D3 <--- DCB address.
    BSR.L      I_UNS_QEVENT          BSR to the routine.
    IF  <NE>   THEN                  If something went wrong,
       TRO$.KILLER ,                 kill the system.
    ENDI
```

HALT I/O Event from Commands (Level 0)

```
    CLR.L      A0                    Event not to be copied.
    MOVE.W     #ISTATO,D1            Time-out message.
    MOVE.L     DCB_ADDR(A5),D3       D3 <--- DCB address.
    BSR.L      N_HLT_QEVENT          BSR to the routine.
    IF  <NE>   THEN                  If something went wrong,
       TRO$.KILLER ,                 kill the system.
    ENDI
```

7.3.2 LOGPHY

The LOGPHY routine converts a logical address to a physical address. LOGPHY also checks to ensure that the address is valid for this task.

7.3.2.1 <u>Entry and Exit Conditions and Register Usage</u>. Entry and exit conditions and register usage for the LOGPHY subroutine are described below.

Subroutine name:

LOGPHY

Entry: D5 = buffer length in bytes
D6 = buffer logical start address (to convert)
A0 = address of TCB of task containing buffer
A5 = address of CCB

Registers usage:

	0	1	2	3	4	5	6	7
Data	S	S	S	S	S	P	P/R	S
Address	P	S	S	S	S	P		

P = parameter register
R = return register
* = destroyed register
S = saved and restored register

Exit: CCR: <EQ> = physical address is in D6.L
<NE> = address is invalid for this task
D6 = physical start address

Exits to: RTS to calling routine

7.3.2.2 <u>LOGPHY Example</u>. In this example, register A5 contains the address of the CCB, and register A3 contains the address of the user's IOCB. The field TCB_ADDR, located in the CCB, holds the physical address of the TCB of the buffer owner.

```
    MOVE.L     IOSSAD(A3),D6        D6 <--- Logical start address.
    MOVE.L     IOSEAD(A3),D5        D5 <--- Logical end address.
    SUB.L      D6,D5
    ADD.L      #1,D5                D5 <--- Buffer length.
    MOVE.L     TCB_ADDR(A5),A0      A0 <--- TCB address.
    BSR.L      LOGPHY               Call LOGPHY.
    IF  <NE>   THEN                 Bad return?
       MOVE.B     #ISTAADD,D1       D1.B <--- Error code.
       BRA        NRM_EXIT          Branch to take care of it.
    ENDI
*
*  At this point, D6 contains the physical start address of the buffer.
*
```

7.3.3 SET_TIME

The SET_TIME routine sets up and performs a TRAP #0 call to generate a wakeup call. The driver writer may want to do this before giving a command to the controller as a time-out, so that the system will not hang waiting for an interrupt that may never come.

This routine will not perform the TRAP #0 call if zero milliseconds were requested.

7.3.3.1 Entry and Exit Conditions and Register Usage.

Subroutine name:

SET_TIME

Entry:
A5 = address of CCB
A3 = address of wakeup routine
D3 = number of milliseconds for wakeup call

Registers usage:

	0	1	2	3	4	5	6	7
Data	S	S	S	P				
Address	S	S		P		P		

P = parameter register
R = return register
* = destroyed register
S = saved and restored register

Exit:
CCR: <EQ> = periodic activation call succeeded
<NE> = periodic activation call did not succeed
or
number of milliseconds = 0

Exits to: RTS to calling routine

7.3.3.2 Notes on Using SET_TIME. The address contained in A3 is where RMS68K will begin execution when the timer goes off. When execution begins, the EXEC saves the contents of registers D0, D1, A0, and A1. (The registers that are normally saved during an interrupt are D0, A0, A1, and A5.)

Register D1 contains an ID that the user identified to the EXEC when the trap call was performed. In this case, the ID is the address of the CCB. So the first thing a user should do after wakeup is push A5 and any other registers that might be used on the stack, and move the contents of D1 into A5. Also, the user must return with an RTE from the wakeup call.

7.3.3.3 SET_TIME Example. In this example, register A5 contains the address of the CCB.

TIMEFLAG is a byte reserved in the CCB to keep track of the status of a wakeup call: if the flag is set, then a wakeup is active and the alternative event has not occurred.

```
*
        LEA       WAKEUP(PC),A3       A3 <--- Address of place to start
*                                     executing upon wakeup.
        MOVE.L    #5000,D3            D3 <--- Number of milliseconds to
*                                     wait for wakeup.
        BSR.L     SET_TIME
*
*  If the wakeup call succeeds, then set TIMEFLAG; otherwise, clear TIMEFLAG.
*
        SEQ.B   TIMEFLAG(A5)
*
*  Send command to device and return control to the system.  If and when the
*  command completes as expected, return to COMM_COMPLETE.
*
*
        BSR.L     SEND_COMMAND

COMM_COMPLETE:
        SF.B      TIMEFLAG(A5)        Clear TIMEFLAG

*
*  Finish the processing ...
*


WAKEUP:

*
*  The system has saved D0, D1, A0, and A1 at this point in processing.
*
RTEREGS REG       D2-D7/A3-A5
        MOVEM.L   RTEREGS,-(SP)       Save registers.
        MOVE.L    D1,A5               D1 contains the request ID that was
*                                     given the EXEC when the periodic
*                                     activation request was set up; the
*                                     request ID is the CCB address.
        TST.B     TIMEFLAG(A5)
        IF  <NE>   THEN               If the flag is set, then the
*                                     alternative event has not occurred,
*                                     and something must be done here.
*
            SF.B   TIMEFLAG(A5)       Clear the flag to show the periodic
*                                     activation request occurred.
*
*    Do whatever processing is needed at this point.
*
        ENDI
        MOVEM.L   (SP)+,RTEREGS
        RTE
```

7.4 TERMLIB ROUTINES

Routines contained in TERMLIB are common for all serial port drivers. These routines include XDEFed routines called as subroutines, background routines called with the BKGRND macro, and transparent mode routines. All are described in the following paragraphs.

7.4.1 XDEFed Routines Called as Subroutines

The externally defined routines that are resident in TERMLIB and called as subroutines are as follows:

LOG_ERR
RESET
TERM_INIT
TERM_COMMAND
TERM_TBE
TERM_BREAK
TERM_GOT_CHAR
TERM_UNRDY
MARK_DOWN

7.4.1.1 LOG_ERR. The LOG_ERR routine logs an error on a received character. A pointer is kept to the last character in the receive queue that had an error. If there was no previous character with an error, then the error code is saved, along with the location of the erroneous character.

Later, when background routine RECV is running, if a character with an error is found an I/O completion event is returned to IOS with the error message.

Entry and Exit Conditions and Register Usage

Entry and exit conditions and register usage for LOG_ERR are described below.

Subroutine name:

LOG_ERR

Entry: A5 = address of CCB
D0.B = error code

Registers usage:

	0	1	2	3	4	5	6	7
Data	P							
Address						P		

P = parameter register
R = return register
* = destroyed register
S = saved and restored register

Exits to: RTS to calling routine

LOG_ERR Example

In this example, the driver is in its interrupt handler, and A5 contains the driver's CCB address.

The user's device sends an interrupt when an error is detected in the process of receiving a character. In this case, the error is a parity error or a framing error.

```
MOVE.B    #ISTACSM,D0          D0.B <--- Error code (from IOE.EQ).
BSR.L     LOG_ERR              Call the LOG_ERR routine.
BRA       INT_EXIT             Branch to the interrupt exit routine.
```

7.4.1.2 RESET. The RESET routine sets and clears flags and pointers in the CCB. It empties the receive and transmit queues and sets the transmit state, receive state, and special flag to idle. If I/O is in progress, RESET sets the menu inactive and decrements the I/O count for the requesting task.

Entry and Exit Conditions and Register Usage

The entry and exit conditions and register usage scheme for the RESET routine are described below.

Subroutine name:

RESET

Entry: A5 = address of CCB

Registers usage:

	0	1	2	3	4	5	6	7
Data								
Address		S				P		

P = parameter register
R = return register
* = destroyed register
S = saved and restored register

Exits to: RTS to calling routine

7.4.1.3 TERM_INIT. This routine is called by the device-dependent module after it receives its initialization call from the CMR handler to perform setup procedures for command service and interrupt service.

TERM_INIT performs the following major functions:

a. Validates the channel type field.

b. Fills in the driver code, recognized attributes, recognized parameters, and recognized baud rate fields in the CCB.

c. Loads the scheduler's address into the CCB for later use.

d. Initializes pointers for the transmit and receive queues.

e. Initializes flags and pointers in the CCB.

f. Sets up the background activation blocks.

References from TCHTYPE Used in TERM_INIT

The file TCHTYPE.AG is assembled at SYSGEN time and linked with TERMLIB.RO to form file TERMLIB.TF. The structure of TCHTYPE is described in paragraph "SYSGENing The New Driver Into The Operating System", later in this chapter.

CH_TYPES	entry point to TCHTYPE table
NUMTYPES	number of channel entries in TCHTYPE table
NUMBYTES	number of bytes in a channel entry in TCHTYPE

Entry and Exit Conditions and Register Usage

The entry and exit conditions and register usage scheme for TERM_INIT are described below.

Subroutine name:

TERM_INIT

Entry: A5 = address of CCB

Registers usage:

	0	1	2	3	4	5	6	7
Data	*	*						
Address	*					P		

P = parameter register
R = return register
* = destroyed register
S = saved and restored register

Exit: CCR: <PL> and <NE> = bad EXEC call
<MI> = channel is down
<EQ> = all went well

Exits to: RTS to calling routine

TERM_INIT Example

In this example, MYINIT is the entry point for the initialization routine for the device-dependent driver.

On entry, register A5 contains the address of the CCB.

DO_INIT is the entry point for the device-dependent initialization routine. Most of DO_INIT can be executed at level 0; this example calls it in call-guarded mode, because some of the data structures accessed by this example are also accessed by some background routines, and those data structures must be protected from simultaneous changes. For more explanation of the background and call-guarded concept, refer to paragraph "Background Routines Called With The BKGRND Macro", later in this chapter, and to Appendix G.

```
MYINIT:
        BSR.L     TERM_INIT           Do the device-independent
*                                     initialization.
        IF  <NE>   THEN               If something went wrong
          IF   <PL>   THEN            and the EXEC call failed,
               TRO$.KILLER ,          call KILLER.
          ELSE
              RTS                     Otherwise, if the channel is
*                                     down, just return.
          ENDI
        ENDI

*
*  Processing reaches this point because all went well with TERM_INIT.
*
        LEA       DO_INIT(PC),A0      Call DO_INIT in call-guarded mode
        MOVE.L    #TOGUARD,D0         to complete the device-dependent
        TRAP      #0                  initialization.
        RTS                           Return to the CMR handler.
```

7.4.1.4 **TERM_COMMAND**. TERM_COMMAND sorts out the command received from the CMR handler and, if necessary, starts I/O.

Entry and Exit Conditions and Register Usage

The entry and exit conditions and register usage scheme for the TERM_COMMAND routine are described below.

Subroutine name:

TERM_COMMAND

Entry:
A2 = physical address of CMR parameter block
A5 = address of CCB
A6 = address of TCB of attached task

Registers usage:

	0	1	2	3	4	5	6	7
Data	*	*				*	*	*
Address	*	*	P	*	*	P	P	

P = parameter register
R = return register
* = destroyed register
S = saved and restored register

Exit: D0 = results of parameter block validation. Always 0 in the driver's case, because IOS will not send any request except HALT I/O if the driver is busy, and the CMR handler makes all the other checks.

Exits to: RTS to calling routine

The calling routine should simply execute an RTS to get back to the CMR handler.

TERM_COMMAND Example

In this example, MYCOMMAND is the entry point for the device-dependent driver's commands. On entry, A2 contains the physical address of the CMR parameter block, and A5 contains the address of the CCB. Also, A6 contains the address of the TCB.

```
MYCOMMAND:

*
*  Perform any necessary device-dependent tasks here, making
*  sure that the contents of A2, A5, and A6 are preserved.
*
         BSR.L     TERM_COMMAND        Call TERM_COMMAND to sort out the
*                                      command and start I/O if necessary
         RTS                           Return to the CMR handler; TERM_COMMAND
*                                      will clear D0 for the return
```

IOS Commands

When TERM COMMAND receives an IOS command (from Input/Output Services), it initiates a series of calls. Table 7-1 shows the correspondence between an IOS command, the TERMLIB routine it calls, the call-guarded routine(s) called by that TERMLIB routine, and the device-dependent routine(s) called by the call-guarded routine(s). The paragraphs that follow describe the sequence of action initiated by each routine.

TABLE 7-1. IOS Command/Routine Hierarchy

IOS COMMAND	TERMLIB ROUTINE	CALL-GUARDED ROUTINES	DEVICE-DEPENDENT ROUTINES
IOHALT	HALT	DO_HALT	
IOWRIT	WRITE	CALL_XMIT	
IOOWIN	OUTWINP	CALL_XMIT	
IOREAD	READ	CALL_XMIT	
IOTBRK	XMIT_BRK	BEG_BREAK	
IOSTAT	REQ_STAT	DO_EVENT	GET_STAT
IOCNFG	CONFIGUR	DO_CONFIGURE	CLOCK_RESET
			SETUP
		DO_EVENT	
IOCHDC	CHNG_DEF	DO_EVENT	GET_STAT

ROUTINES CALLED BY TERM_COMMAND

The following routines are called by TERM_COMMAND on receipt of an IOS command.

HALT

The TERMLIB routine HALT is called when an IOHALT command is received from IOS. It performs the following steps:

a. Calls DO_HALT in a call-guarded manner.

b. Branches to CMD_EXIT.

WRITE

The TERMLIB routine WRITE is called when an IOWRIT command is received from IOS. It performs as follows:

a. Calls IO_COMN.

b. If the driver is requested to clear the discard output flag, WRITE clears it and sets INHIB_DO.

c. Sets up the menu for the type of write requested and the END directive, so that a completion event will be sent when the write finishes.

d. Branches to START_IO.

OUTWINP

The TERMLIB routine OUTWINP is called when an IOOWIN command is received from IOS. It does the following:

a. Calls IO_COMN.

b. Calls BUF_ADDR to obtain read buffer physical address.

c. Calls RD_COMN.

d. Sets up menu according to requested read and write, and appends the END directive.

e. Branches to START_IO.

READ

The TERMLIB routine READ is called when an IOREAD command is received from IOS. It performs as follows:

a. Calls IO_COMN.

b. Calls RD_COMN.

c. Sets up menu with type read requested; sets up the END directive.

d. Branches to START_IO.

XMIT_BRK

The TERMLIB routine XMIT_BRK is called when an IOTBRK command is received from IOS. It accomplishes the following:

a. Calls BEG_BREAK in a call-guarded manner. This is the entry point used when the background routine B_BRK is called as a subroutine.

b. Branches to CMD_EXIT.

REQ_STAT

The TERMLIB routine REQ_STAT is called when an IOSTAT command is received from IOS. It performs the following actions:

a. Calls the device-dependent routine GET_STAT.

b. Fills in the status, type, attributes word, and parameter fields of the user's Configure/Status Block (CSB).

c. Copies the configuration from the CCB to the user's CSB.

d. Branches to NRM_EXIT.

CONFIGUR

The TERMLIB routine CONFIGUR is called when an IOCNFG command is received from IOS. It does the following:

a. Copies the time-out values from the DCB to the CCB.

b. Calls CMN_CNFG.

c. Calls DO_CONFIGURE in a call-guarded manner.

d. Branches to NRM_EXIT.

CHNG_DEF

The TERMLIB routine CHNG_DEF is called when an IOCHDC command is received from IOS. It performs the following steps:

a. Calls CMN_CNFG.

b. If user is requesting a change to transparent mode now, CHNG_DEF installs user's branch table address in the CCB at HIS_BRA_TABL, writes the address of the driver's jump table in the user's CSB, and also writes to the CSB the contents of A5 and the driver's interrupt level.

c. Calls the device-dependent routine GET_STAT.

d. Copies the new default configuration into the CCB.

e. Branches to NRM_EXIT.

ROUTINES CALLED AS SUBROUTINES

The following routines are called as subroutines in the command sequence initiated by TERM_COMMAND on receipt of an IOS command.

IO_COMN

The IO_COMN subroutine is called by TERMLIB routines WRITE, OUTWINP, and READ. It performs the following actions:

a. Calls the device-dependent routine GET_STAT.

b. If the device is not ready, sets up the proper error code and branches to NRM_EXIT.

c. If the device is ready, but the driver is busy detecting auto baud rate, IO_COMN sets up the error code and branches to NRM_EXIT.

d. If the device is ready and the driver is not in the middle of detecting baud rate, IO_COMN checks to see if the IO buffer is in a different task than that of the IOCB.

e. If the IO buffer is in a different task than that of the IOCB, then IO_COMN calls the TRAP #0 call GETTCB. If the call is good, it puts the physical address of the TCB into the CCB. If the call is bad, it sets the status in the IOCB and branches to NRM_EXIT.

f. Clears XFER_LEN.

g. Calls BUF_ADDR to get write buffer physical address.

h. Returns (with an RTS instruction) to the routine that called it.

CMN_CNFG

The CMN_CNFG subroutine is called by both the TERMLIB routines CONFIGUR and CHNG_DEF. It performs as follows:

a. Obtains the physical address of the CSB.

b. If the attributes mask or parameters mask is not legal, CMN_CNFG sets up the proper error code, puts the ISTACNF code in the status area, and branches to NRM_EXIT.

c. If the attributes mask and the parameters mask are acceptable, then CMN_CNFG copies the proposed configuration into the CSB.

d. If any parameter is illegal, CMN_CNFG sets the corresponding error code, puts ISTACNF in the status area, and branches to NRM_EXIT.

e. Otherwise, if all the fields are legal, CMN_CNFG sets up the fields at the top of the CSB, sets the DCBCCF flag, and returns to the routine that called it.

DO_HALT

Call-guarded subroutine DO_HALT is called by TERMLIB routine HALT. It does the following:

a. If I/O is being performed, DO_HALT puts the time-out status into the event area, sets the status field of the IOCB with the time-out code, sets the length of the data transfer, and calls DDP_RESET.

b. If I/O is not being performed, DO_HALT puts the abort status into the event area.

c. In either case, DO_HALT sends an event by calling Q_EVENT.

d. If the event is sent without error, DO_HALT returns to HALT.

e. If errors occur in sending the event, DO_HALT proceeds to KILLER.

RD_COMN

The RD_COMN subroutine is called by TERMLIB routines OUTWINP and READ. It does the following:

a. Sets the INPUT and INHIB_DO flags. Clears DISCARD, CHAR_CNT, and ECHO.

b. If the options word says that the driver should not suppress echo, but the driver is configured to echo, then RD_COMN sets ECHO.

c. Returns to the TERMLIB routine that called it.

BUF_ADDR

The BUF_ADDR subroutine is called by the TERMLIB routine OUTWINP and by the IO_COMN subroutine. It performs as follows:

a. Calls GET_PHYSICAL_ADDR.

b. If there was an error while getting the physical address, BUF_ADDR sets up the error code and branches to NRM_EXIT.

c. Otherwise, BUF_ADDR sets up the physical start, end, and length in registers D4, D5, and D6, and executes an RTS instruction.

GET_PHYSICAL_ADDR

The GET_PHYSICAL_ADDR subroutine is called by the BUF_ADDR subroutine. It does the following:

a. Calls the externally referenced (XREFed) routine LOGPHY.

b. Returns to BUF_ADDR.

CALL_XMIT

The call-guarded subroutine CALL_XMIT is called by START_IO, and by TERMLIB routines WRITE, OUTWINP, and READ. CALL_XMIT accomplishes the following:

a. Obtains the device address from register A1 and the driver address from A4; sets the NONINTERRUPT flag.

b. Calls XMIT.

c. Clears the NONINTERRUPT flag.

d. Executes an RTS instruction.

DO_EVENT

The call-guarded subroutine DO_EVENT is called by NRM_EXIT. It performs the following steps:

a. Sets the NONINTERRUPT flag.

b. Loads the address of the event area into A0.

c. Calls Q_EVENT.

d. Clears the NONINTERRUPT flag.

e. If there was a bad return from Q_EVENT, goes to KILLER.

f. Executes an RTS instruction.

Q_EVENT

The Q_EVENT subroutine is called by the call-guarded routine DO_HALT and the call-guarded subroutine DO_EVENT. Q_EVENT performs as follows:

a. Saves the registers the driver will destroy.

b. Depending on the type of event the driver wants to send, and whether or not the NONINTERRUPT flag is set, Q_EVENT calls the appropriate routine from the QEVENT module.

c. Restores the registers and executes an RTS instruction.

ROUTINES USED AS EXIT POINTS

The following routines are used as exit points for the command sequence initiated by TERM_COMMAND on receipt of an IOS command.

START_IO

TERMLIB routines WRITE, OUTWINP, and READ branch to START_IO, which does the following:

a. Increments outstanding I/O count in user's TCB.

b. Calls CALL_XMIT in a call-guarded manner.

c. Branches to CMD_EXIT.

NRM_EXIT

The TERMLIB routines REQ_STAT, CONFIGUR, and CHNG_DEF; and the subroutines CMN_CNFG, IO_COMN, and BUF_ADDR; branch to NRM_EXIT, which performs the following actions:

a. Puts the status into the IOCB.

b. Prepares the completion event.

c. Calls DO_EVENT in a call-guarded manner.

d. Branches to CMD_EXIT.

CMD_EXIT

TERMLIB routine XMIT_BRK, and the START_IO and NRM_EXIT routines, branch to CMD_EXIT, which performs as follows:

a. Moves a 0 into register D0 to indicate all is well.

b. Restores the stack pointer saved at the beginning of commands.

c. Returns (with an RTS) to the CMR handler.

7.4.1.5 TERM_BREAK. The TERM_BREAK routine is called by the device-dependent module when it receives a break signal. If the driver is in transparent mode, this routine calls the transparent mode break routine and exits. Otherwise, TERM_BREAK calls the background BREAK routine.

Entry and Exit Conditions and Register Usage

The entry and exit conditions and register usage scheme for the TERM_BREAK routine are described below.

Subroutine name:

TERM_BREAK

Entry: A5 = address of CCB

Interrupt level: INHIBITED. Called by interrupt handler.

Registers usage:

	0	1	2	3	4	5	6	7
Data								
Address	*					P		

P = parameter register
R = return register
* = destroyed register
S = saved and restored register

Exits to: RTS to calling routine

TERM_BREAK Example

In this example, the device-dependent driver has received an interrupt which notifies it that it has just received a break signal. Also, register A5 contains the address of the CCB.

```
BSR.L     TERM_BREAK          Call the TERM_BREAK routine.
BRA.L     INT_EXIT            Branch to the interrupt exit routine.
```

7.4.1.6 **TERM_TBE**. The TERM_TBE routine is called by the device-dependent module when a transmit-buffer-empty interrupt has been received. If the driver is in the transparent mode, TERM_TBE calls the transparent mode transmit-buffer-empty routine. Otherwise, it calls background routine XMIT.

Entry and Exit Conditions and Register Usage

The entry and exit conditions and register usage scheme for the TERM_TBE routine are described below.

Subroutine name:

TERM_TBE

Entry: A5 = address of CCB

Interrupt level: INHIBITED. Called by interrupt handler.

Registers usage:

	0	1	2	3	4	5	6	7
Data								
Address	*					P		

P = parameter register
R = return register
* = destroyed register
S = saved and restored register

Exits to: RTS to calling routine

TERM_TBE Example

In this example, the device-dependent driver has just received a transmit-buffer-empty interrupt, register A5 contains the address of the CCB, and all device-dependent processing required has been done.

```
        BSR.L     TERM_TBE            Call TERM_TBE.
        BRA.L     INT_EXIT            Branch to interrupt exit routine.
```

7.4.1.7 **TERM_GOT_CHAR**. The TERM_GOT_CHAR routine is called by the device-dependent module when it receives a character. If the driver is in transparent mode, the driver calls the user's received-character-available routine and executes an RTS instruction.

If the driver is not in transparent mode, it looks for special characters listed below and responds as described.

a. Not NUL and BREAK equivalent:
TERM_GOT_CHAR calls background BREAK routine and exits.

b. Not BREAK equivalent, the driver is blocked, and any character is XON:
TERM_GOT_CHAR calls background routine UNBLK and exits.

c. NUL:
If the driver is not passing nulls as data, TERM_GOT_CHAR ignores the character and exits. Otherwise, it treats character as NON_SPECIAL.

d. XON:
If blocked, TERM_GOT_CHAR calls background routine UNBLK and exits.

e. XOFF:
TERM_GOT_CHAR sets BLOCKED flag, calls background routine BLOCK, and exits.

f. DISCARD OUTPUT character:
If the discard output function is not currently inhibited, the TERM_GOT_CHAR routine inverts the DISCARD flag. Otherwise, it ignores the character and exits.

g. NOT_SPECIAL:
TERM_GOT_CHAR tries to put the character in the receive queue. If the receive queue is full, it logs an overrun error and exits. If near full, it calls background routine STOP, calls background RECV, and exits.

Entry and Exit Conditions and Register Usage

The entry and exit conditions and register usage scheme for TERM_GOT_CHAR are described below.

Subroutine name:

TERM_GOT_CHAR

Entry: A5 = address of CCB
D0.B = the character received

Interrupt level: INHIBITED. Called by interrupt handler.

Registers usage:

	0	1	2	3	4	5	6	7
Data	P	*						
Address	*					P		

P = parameter register
R = return register
* = destroyed register
S = saved and restored register

Exits to: RTS to calling routine

TERM_GOT_CHAR Example

In this example, register A5 contains the address of the CCB, and an interrupt has occurred which tells the device-dependent driver that a character has been received. The device-dependent driver has put the character into D0.B. All device-dependent processing has been done.

```
BSR.L     TERM_GOT_CHAR        Call TERM_GOT_CHAR.
BRA.L     INT_EXIT             Branch to the interrupt exit routine.
```

7.4.1.8 TERM_UNRDY. The TERM_UNRDY routine is called by the device-dependent module when it discovers that the terminal status has changed to unready. If the driver is performing I/O, it sends a completion event with a not ready status and calls RESET and DDP_RESET.

Entry and Exit Conditions and Register Usage

The entry and exit conditions and register usage scheme for TERM_UNRDY are described below.

Subroutine name:

TERM_UNRDY

Entry: A5 = address of CCB

Interrupt level: Level 0. Called in background.

Registers usage:

	0	1	2	3	4	5	6	7
Data								
Address	*	*			S	P		

P = parameter register
R = return register
* = destroyed register
S = saved and restored register

Exit: CCR: <PL> and <NE> = bad EXEC call
<MI> = channel is down
<EQ> = all went well queuing the event, or driver was not doing I/O

Exits to: RTS to calling routine

TERM_UNRDY Example

In this example, register A5 contains the address of the CCB. The device-dependent driver has received an interrupt which tells it that the terminal status has changed to unready. It has called a background routine to take care of this condition, so the driver is executing at interrupt level 0.

```
BSR.L     TERM_UNRDY
IF  <NE>   THEN
   IF  <PL>  THEN                TERM_UNRDY tried to queue an
    TRO$.KILLER ,                event and the EXEC call failed,
   ENDI                          so kill the system.
ENDI
RTS
```

7.4.1.9 MARK_DOWN. The MARK DOWN routine is called by the device-dependent module when it detects that the channel is down because it is not ready. The driver sets MENU to M_DOWN, and moves ISTAUNR into DOWN_ERR.

Entry and Exit Conditions and Register Usage

The entry and exit conditions and register usage scheme for MARK_DOWN is described below.

Subroutine name:

MARK_DOWN

Entry: A5 = address of CCB

Interrupt level: Level 0. Called in background.

Registers usage:

	0	1	2	3	4	5	6	7
Data								
Address						P		

P = parameter register
R = return register
* = destroyed register
S = saved and restored register

Exits to: RTS to calling routine

MARK_DOWN Example

In this example, register A5 contains the address of the CCB. In DO_INIT, the device-dependent driver's initialization routine, the driver wants and is able to do a diagnostic test and, if the test fails, wants to mark the channel down. The driver is executing in the background at interrupt level 0.

```
DO_INIT:
*
*  Inhibit interrupts
*

         MOVE.L    SR,-(SP)
*
*  Perform the device-dependent tasks ...
*


*
*  Test the device ...
*
*  If the device fails, then

         BSR.L     MARK_DOWN
*
*  Restore interrupt level
*
         MOVE.L    (SR)+,SR
         RTS
```

7.4.2 Background Routines Called with the BKGRND Macro

The background routines available in TERMLIB that can be called with the BKGRND macro are

RECV
BREAK
XMIT
BLOCK
UNBLK
STOP
B_BRK
E_BRK

Most of these routines are called only by TERMLIB, not by the device-dependent module. A routine that is called by the device-dependent module is BREAK.

7.4.2.1 How the Background and Call-Guarded Modes Work. Using the background routines allows a driver to kick off routines that can run at interrupt level 0. This is desirable because running at level 0 allows other devices at the same or lower interrupt level to use the processor.

Only one background or call-guarded routine is allowed to run at a time. This provides a mutually exclusive relationship to protect data structures used in two background routines. If a background routine uses a data structure that can be modified by a routine running as a result of an interrupt, the background routine must enable/unmask around the critical section to protect the data structure from corruption.

Appendix G contains details about the EXEC background and call-guarded modes.

7.4.2.2 Using the BKGRND and SET_BAB Macros. Macros that help the driver writer set up an environment for using the background routines are found in file 9995..UTILITY.MC. These macros simplify the process described in Appendix G by performing the functions of the BKG_SCHEDULE and ENTRY macros.

SET_BAB Macro

The Background Activation Block (BAB) is illustrated in Figure 7-1.

```
           +----+----+----+----+
$ 0 BABLNK |    |    |    |    |---------> Forward link to next BAB in queue
           +----+----+----+----+
$ 4 BABPTR |    |    |    |    |---------> Pointer to routine to be
           +----+----+----+----+           executed when background runs
$ 8 BABUSE |    |                ---------> Flag used by EXEC: true (nonzero)
           +----+----+----+                means BAB in use
$ 9        |    |    |    |      ---------> Reserved (should be zero)
           +----+----+----+

 BABBLN = $C = length of a BAB
```

FIGURE 7-1. Background Activation Block Structure

The SET_BAB macro requires two arguments: the label for the start of the routine to be executed when the background mode runs, and the prefix (name) of the label for the BAB (where the label for the BAB is of the form <name>_BAB.)

This macro installs the address of the routine to be executed when the background mode runs into the BABPTR field of the BAB. TERMLIB uses this macro in TERM_INIT to set up its BABs. The device-dependent module should also use the SET_BAB macro during its device-dependent initialization to set up any device-dependent BABs. Figure 7-2 illustrates an example of the SET_BAB macro.

```
Call:
         SET_BAB     RECV0,RECV

Result:
         A pointer to RECV0 is installed in the RECV_BAB BAB

                   +----+----+----+----+
RECV_BAB:          |    |    |    |    |
                   +----+----+----+----+
                   |    |    |    |    |---------> RECV0:  to be executed
                   +----+----+----+----+           when background mode runs
                   |    |
                   +----+----+----+
                   |    |    |    |
                   +----+----+----+
```

NOTE: The SET_BAB macro destroys the contents of register A0.

FIGURE 7-2. SET_BAB Macro Example

BKGRND Macro

The BKGRND macro moves the BAB address from the CCB into address register A0, and then executes a JSR to the address of the background scheduler routine in EXEC (provided by the ENTRY call).

The BKGRND macro requires two arguments. The first is the prefix (name) of the label (<name>_BAB) of the BAB. The second is the label of the offset in the CCB where the address of the EXEC's scheduler is found; the default is SCHED_EP. This address is installed in the CCB during initialization.

The scheduler's address is obtained through a TRAP #0 call. TERMLIB uses the BKGRND macro to call its background routines after initialization. A routine already running in background mode can also call a background routine, including itself. The device-dependent module should use the BKGRND macro to call any background routines that must be called.

Call:

```
        BKGRND    RECV
```

Result:

The background routine pointed to by BABPTR in the RECV_BAB background activation block will execute at some time in the future.

7.4.2.3 How to Write a Background Routine.

When a BAB has been set up, the BABPTR offset is pointing to some code, and the background routine is called, driver writers know that A1 contains the address of the BAB. However, they must find their CCB address. Because their BAB is in the CCB, they can take a negative offset from A1 to discover the CCB address.

For example, in the RECV background routine, driver writers should do the following:

```
RECV0:

        LEA       -RECV_BAB(A1),A5
        MOVE.L    CCBCHB(A5),A1
        MOVE.L    DRV_ADDR(A5),A4
```

They are now set up with their CCB address in A5, their device-dependent address in A1, and the address of their device-dependent module in A4. They are free to destroy any registers they choose.

7.4.2.4 RECV.

The RECV background routine is not called directly by the device-dependent module. It is called from the following three locations:

a. From DISPATCH when the driver has a command to perform a formatted or image mode read.

b. From the background routine XMIT when the transmit state is idle and the WAIT_TQ flag is set.

c. From the routine TERM_GOT_CHAR, which is called because the driver has received a receive character interrupt from the device, the character is not special, and the receive queue is not full.

The RECV background routine does the following:

a. Checks the RECV_ST, RQ_CNT, and TQ_CNT flags to make sure the driver has a character to receive and that there is room in the transmit queue. If there is room in the transmit queue, RECV clears the WAIT_TQ flag; otherwise, it sets the flag.

b. Checks to see if there is an error on the character.

c. If the driver passes all these tests, then RECV retrieves the character from the receive queue.

d. If the driver was stopped but there is enough room in the receive queue after this character is removed, the driver reactivates the device.

e. Then the driver jumps to the proper routine according to its receive state. It could be idle, reading in image mode, or reading in formatted mode.

f. If the driver is idle, it just returns (RTS).

g. If the driver is reading in image mode, it checks for room in the user's buffer, echoing, and termination characters. If putting this character in fills the user's buffer and the driver is configured to terminate when the buffer is full, then the driver clears the receive state, calls XMIT in the background, and returns (RTS). Otherwise, it calls TERMTST and then branches to RECV to try again.

h. If the driver is in format mode, the situation is more complicated. If the character is a terminating character, the driver sets up to transmit an end-of-line string, clears the receive state, calls the background routine XMIT, and returns (RTS). If the character is a NUL or non-ASCII, it sets up to transmit a BEL and then branches to RECV to try again. If the character is a DEL or a backslash, the driver performs some logic to remove the character and then branches to RECV. If the character is a reprint line character, the driver sets up to transmit end-of-line, changes the menu to "reprint", clears the receive state, and calls background routine XMIT.

   If the character is the cancel line character, and the driver is echoing and is not a hardcopy device, the driver loads CANCEL into the menu, clears the receive state, and calls background routine XMIT.

   If the character is the cancel line character, and the driver is not echoing and is not a hardcopy device, the driver sets up to transmit a backslash and end-of-line string, calls background routine XMIT, and branches to RECV.

If the character is ordinary data, and the RBUF is not full, the driver loads the character in the RBUF user's buffer. If the driver is echoing, it takes care of that. If the driver is configured to terminate when the RBUF is full and it is full, the driver sets up the transfer length, sets up to transmit the end-of-line string, clears the receive state, calls background routine XMIT, and returns (RTS). If the driver is not configured to terminate when the buffer is full, or if the buffer is not full, the driver branches to RECV.

7.4.2.5 BREAK. The BREAK background routine is called by the TERM_GOT_CHAR routine and the TERM_BREAK routines in TERMLIB. BREAK is also called by device-dependent routines when a break is detected.

BREAK works in the following way:

a. If the driver is not ready to report breaks (i.e., if no command has been received), is in transparent mode, or is down or inactive, BREAK just returns (RTS).

b. Otherwise, if the driver is not performing I/O, BREAK sends an unsolicited device event. If I/O is being performed, BREAK sends a completion event with the ISTABRK error code.

c. BREAK then resets the device.

7.4.2.6 XMIT. XMIT is frequently called from other routines in TERMLIB. TERM_TBE calls XMIT when a transmit-buffer-empty interrupt is received. The background routines RECV, UNBLK, STOP, and E_BRK call XMIT, as do the subroutines TERM_TST, DISPATCH, UNSTOP, Q4_XMIT, and Q_XMIT. Also, when a command has been received and the driver wants to start I/O, it calls XMIT.

The XMIT background routine does the following:

a. If the transmit buffer is not empty, XMIT just returns (RTS).

b. Otherwise, XMIT jumps to one of the routines corresponding to the transmit state (XMIT_ST). These routines are

1.	X_IDLE	Clear transmit state.
2.	X_IMAGE	Write in image mode.
3.	X_FORMAT	Write in formatted mode.
4.	X_CANCEL	Cancel a line.
5.	X_REPRNT	Reprint a line.

6.	X_SPECIAL	The SPECIAL word contains bits which correspond to the following routines. Save the old transmit state and perform one of these for a while.
(a)	X_BREAK	NOP
(b)	X_XOFF	Clear SPEC_XOFF bit in SPECIAL, put out the XOFF character.
(c)	X_XON	Clear SPEC_XON bit in SPECIAL, put out the XON character.
(d)	X_BLOCK	NOP
(e)	X_NUL	If NUL_CNT is not zero, subtract 1 from NUL_CNT and put out NUL character. If NUL_CNT is zero, clear the SPEC_NUL bit in SPECIAL.
(f)	X_QUEUE	If TQ_CNT is not zero, remove a character from the transmit queue. Put out the character. If TQ_CNT is zero, clear the SPEC_QUEUE bit in SPECIAL.

ROUTINES CALLED IN XMIT EXECUTION

The following routines, with the exception of DISPATCH and X_F_TERM, are called by the XMIT routine. The transmit state (XMIT_ST) determines which routine is called. X_IDLE branches to the DISPATCH subroutine, which calls XMIT. X_FORMAT branches to the X_F_TERM background routine, which calls XMIT.

X_IDLE

The X_IDLE background routine performs as follows:

a. If the driver had a task to do the last time the RECV background routine was called, but the transmit queue was too full (WAIT_TQ not 0), then X_IDLE calls background routine RECV.

b. If RECV was not waiting on the transmit queue, and the receive state is not IDLE, then X_IDLE returns (RTS).

c. If RECV was not waiting on the transmit queue and the receive state is IDLE, then X_IDLE branches to the DISPATCH subroutine and returns (RTS).

DISPATCH

DISPATCH reads the next item to determine what type of I/O to do next. If the driver is down or inactive, DISPATCH returns (RTS).

If it is a transmit-type item, DISPATCH sets up the transmit state to correspond to the state of the item, and calls background routine XMIT. Transmit-type items are the following:

a.	S_IDLE	Idle
b.	S_IMAGE	Do image write
c.	S_FORM	Do format write
d.	S_CANCEL	Cancel line
e.	S_REPRNT	Reprint line
f.	S_SPECIAL	Do a special routine

If it is a receive-type item, DISPATCH clears LAST_BS and sets the receive state to correspond to the menu item. If necessary, it clears the type-ahead buffer. It then calls background routine RECV. Receive-type menu items are the following:

a.	S_IDLE	Idle
b.	S_IMAGE	Image read
c.	S_FORM	Formatted read

If it is neither a transmit-type menu item nor a receive-type menu item, then the driver assumes an end menu item and sends a completion event to IOS. It then returns (RTS).

X_IMAGE

X_IMAGE performs the following actions:

a. If the driver is past the end of the WBUF buffer, X_IMAGE sets up the transfer length (if a write was in process), clears the transmit state, and branches to X_IDLE.

b. Otherwise, it puts the character into D0 and branches to XMIT_D0.

X_FORMAT

The X_FORMAT routine performs as follows:

a. If the driver is past the end of the WBUF buffer, or if the character in the buffer is a carriage return, X_FORMAT branches to X_F_TERM.

b. Otherwise, it updates the WBUF pointer. If the character is printable, X_FORMAT adds 1 to the column count. It then branches to XMIT_D0.

X_F_TERM

The X_F_TERM background routine does the following:

a. Clears the transmit state, and sets up the transfer length if the driver is executing a Write command.

b. If driver is configured with either a zero or positive line length, and the number of printable characters is not an exact multiple of the line length, X_F_TERM puts the end-of-line string into the transmit queue and branches to subroutine Q4_XMIT.

c. It then branches to XMIT.

X_CANCEL

The X_CANCEL background routine performs according to the following rules:

a. If character count is zero, X_CANCEL clears the transmit state and branches to X_IDLE.

b. Otherwise, it subtracts 1 from the character count; sets up to transmit backspace, space, backspace, by branching to subroutine Q4_XMIT; and then returns (RTS).

X_REPRNT

The X_REPRNT background routine executes the following sequence:

a. If everything has been reprinted, X_REPRNT clears the transmit state and branches to X_IDLE.

b. Otherwise, it locates next character from the WBUF buffer and adds 1 to the character count. If this is a control character, X_REPRNT puts a caret symbol (^) followed by the character's visible counterpart into the transmit queue using Q4_XMIT; it then returns.

c. If this is not a control character, X_REPRNT puts the character into D0 and branches to XMIT_D0.

X_SPECIAL

The X_SPECIAL background routine tests bits in the SPECIAL word to determine which task to do. Requested tasks are done in the following order:

X_BREAK
X_XOFF
X_XON
X_BLOCK
X_NUL
X_QUEUE

If none of the specified bits are set in the SPECIAL word, then the previous transmit state is restored and the driver branches to XMIT_JMP, which dispatches to one of the transmit state routines.

X_BREAK

NOP

X_XOFF

The X_XOFF task clears the X_XOFF bit in the SPECIAL word and sends the XOFF character by branching to FORCE_DO.

X_XON

The X_XON task clears the X_XON bit in the SPECIAL word and sends the XON character by branching to FORCE_DO.

X_BLOCK

NOP

X_NUL

If NUL_CNT is zero, the X_NUL task clears the bit for X_NUL and branches to X_SPECIAL. Otherwise, it puts NUL in D0, subtracts 1 from NUL_CNT, and branches to XMIT_D0.

X_QUEUE

If transmit queue count is zero, then the X_QUEUE task clears the bit for X_QUEUE and branches to X_SPECIAL. Otherwise, it gets the next character from the transmit queue and branches to XMIT_D0.

XMIT_DO

If the driver is discarding output, the XMIT_DO task then clears NUL_CNT, empties the transmit queue, and moves the WBUF buffer pointer past the end of the buffer. It clears either the previous transmit state or the current transmit state, depending on whether or not the driver is in special mode. Then XMIT_DO branches to XMIT.

If the driver is not discarding output, then XMIT_DO puts the character in DO. If it is a carriage return or a line feed, and the driver is configured for null padding, XMIT_DO sets the bit for X_NUL in SPECIAL. Then it returns (RTS).

SUBROUTINES USED IN XMIT EXECUTION

The subroutines described in the following paragraphs are often used during execution of the XMIT background routine.

Q4_XMIT

The Q4_XMIT subroutine is called by routines that must put up to four characters in the transmit queue, set up the state as special, set the bit for X_QUEUE, and call background routine XMIT.

Q_XMIT

The Q_XMIT subroutine is called by routines that must put one character in the transmit queue, set up the state as special, set the bit for X_QUEUE, and call background routine XMIT.

SET_SPECIAL

The SET_SPECIAL subroutine is called by routes that must, if the driver is not running something special, save the current transmit state and change the current transmit state to special.

7.4.2.7 BLOCK. The background routine BLOCK is called by the subroutine TERM_GOT_CHAR when the received character is XOFF.

7.4.2.8 UNBLK. The background routine UNBLK is called by the subroutine TERM_GOT_CHAR when the driver is BLOCKED and the received character is XON, or when the driver is configured to accept any character as an XON.

7.4.2.9 __STOP__. Background routine STOP is called by subroutine TERM_GOT_CHAR when the received character is not a special character, the driver must put it into the receive queue, and the receive queue is nearly full.

7.4.2.10 __B_BRK__. B_BRK is never called as a background routine in TERMLIB; instead BEG_BREAK, its subroutine entry point, is called in a call-guarded manner by XMIT_BRK, the command from IOS that requests the driver to transmit a break. BEG_BREAK calls the device-dependent subroutine DDP_BEG_BREAK to send the break signal, sets up for execution of a special routine, sets the bit for BREAK in the SPECIAL word, and uses a periodic activation call to set up a timer that will start execution at the routine STOP_BRK.

7.4.2.11 __STOP_BRK__. The STOP_BRK background routine performs as follows:

a. If the driver is in transparent mode, STOP_BRK calls DDP_END_BREAK to prevent the device from sending the break. Then it jumps (JSR) to the user's break-is-over routine.

b. If the driver is not in transparent mode, STOP_BRK calls background routine E_BRK to stop the break signal.

c. RTE from the timer interrupt.

7.4.2.12 __E_BRK__. The E_BRK background routine is called by STOP_BRK when the periodic activation call has notified the driver that it is time to end a break.

7.4.3 Transparent Mode Routines as Implemented in TERMLIB

Transparent mode, which is described in the following paragraphs, is used by the CONNECT utility.

7.4.3.1 __How Transparent Mode Is Set Up__. With the driver's transparent mode bit (15) set in the attributes word and attributes mask, a Configure-Defaults command is received from IOC. The driver sets the flag TR_MODE, gets the address of the transparent mode branch table from the Configure/Status Block (CSB), validates and converts the address from logical to physical, and saves it in HIS_BRA_TABL in the CCB.

Then the driver exchanges the address of its transparent mode branch table for the address the user designated.

The driver then writes to the CSB the contents of its A5 register and the interrupt level at which it must be entered.

The user's CSB on entry to the Configure-Defaults command is diagrammed in Figure 7-3.

```
    user's     +-----------------+
    CSB -----> |                 |
               +-----------------+
               |                 |
                        .
                        .
                        .
               +-----------------+      +-------------------+
     $2A       |user's bra table |----> |                   |---> TM_RCA
               +-----------------+      +-------------------+
     $2E       |                 |      |                   |---> TM_TBE
               +-----------------+      +-------------------+
     $32       |        |               |                   |---> TM_BRK_RECV
               +--------+               +-------------------|
                                        |                   |---> TM_BRK_OVER
                                        +-------------------+
```

FIGURE 7-3. User CSB Entering Configure-Defaults Command

The user's CSB on exit from the Configure-Defaults command is diagrammed in Figure 7-4.

```
    user's     +-----------------+
    CSB -----> |                 |
               +-----------------+
               |                 |
                        .
                        .
                        .
               +-----------------+      +-------------------+
     $2A       |driver bra table |----> |                   |---> TM_OUTPUT
               +-----------------+      +-------------------+
     $2E       |driver A5        |      |                   |---> TM_BREAK
               +-----------------+      +-------------------+
     $32       |int lvl|
               +-------+
```

FIGURE 7-4. User CSB Exiting Configure-Defaults Command

After the driver is configured for transparent mode, the user can directly call the TM_OUTPUT and TM_BREAK routines. The only commands allowed from IOS are IOSTAT (Status) and IOCHDC (Change-Default-Configuration). Also, some driver routines call user routines, as follows:

TERM_GOT_CHAR	calls transparent mode TM_RCA
TERM_TBE	calls transparent mode TM_TBE
TERM_BREAK	calls transparent mode TM_BRK_RECV
STOP_BRK	calls transparent mode TM_BRK_OVER

Background routine BREAK does nothing if the driver is in transparent mode.

7.4.3.2 TM_OUTPUT. Background routine TM_OUTPUT calls the device-dependent routine PUT_CHAR to output the character in D0, sets the Condition Code Register (CCR) to <EQ>, and returns (RTS).

7.4.3.3 TM_BREAK. The background routine TM_BREAK disables interrupts, causes the device to send a break by calling DDP_BEG_BREAK, and sets up a periodic activation call so that, when the timer goes off, execution will begin at STOP_BRK. It then sets the CCR to <EQ> and returns (RTS).

7.5 WRITING THE DEVICE-DEPENDENT MODULE

In most ways, the device-dependent module looks like other drivers. However, a branch table to device-dependent routines must be installed near the start, immediately after the required service vector table, at offset $28. Also, because DRVLIB and TERMLIB are in sections 15 and 14, respectively, code in the device-dependent module should be in a different section, such as 0 or 8.

7.5.1 Tables and Routines Required by TERMLIB

The following paragraphs describe the tables and routines required by TERMLIB.

7.5.1.1 Branch Table. Users should place the following branch table at offset $28 from the beginning of the driver.

```
BRA.L       PUT_CHAR
BRA.L       CK_TBE
BRA.L       DDP_RESET
BRA.L       SETUP
BRA.L       CLOCK_RESET
BRA.L       GET_STAT
BRA.L       DDP_STOP
BRA.L       DDP_UNSTOP
BRA.L       DDP_BEG_BREAK
BRA.L       DDP_END_BREAK
```

7.5.1.2 Initialization Requirements. Device-independent initialization will be performed in TERM_INIT, and device-dependent initialization should be done in a DO_INIT subroutine, which should be called in a call-guarded manner:

```
LEA      DO_INIT(PC),A0
MOVE.L   #T0GUARD,D0
TRAP     #0
```

DO_INIT must accomplish the following:

a. Load starting address of the driver into the CCB at offset DRV_ADDR. TERMLIB uses this address to jump to the device-dependent routines.

   For example, if the label at the start of a driver is MYDRVR, then the following commands should be in DO_INIT:

```
LEA      MYDRVR(PC),A0
MOVE.L   A0,DRV_ADDR(A5)
```

b. Set up the BABs for any device-dependent background routines used by a driver.

   If users have a device-dependent background routine, they must have reserved 12 bytes for the BABs in the device-dependent part of the CCB, following the offset TERMDDP.

   The label on a BAB should be xxxx_BAB, where xxxx is chosen by the user. For example, if users need a background routine for a DSR interrupt, they allocate storage in the CCB as follows, using the RESERVE macro from file 9995..UTILITY.MC:

```
OFFSET   TERMDDP
RESERVE.12  DSR_BAB
```

   In DO_INIT, users must load the addresses of the background routine into the BAB. If the entry point for the DSR background routine is DSR0, then the SET_BAB macro must be used:

```
SET_BAB   DSR0,DSR
```

   The DSR0 address will be installed in the DSR_BAB BAB.

   Care is required because the SET_BAB macro destroys the contents of register A0.

   Users may now perform any necessary device-dependent initialization.

7.5.1.3 PUT_CHAR: Put Out a Character to the Device. PUT_CHAR writes the character in D0.B to the device. This routine is called by XMIT_DO (FORCE_DO) and TM_OUTPUT in TERMLIB. Entry and exit conditions are described below.

Entry: A5 = address of CCB
A4 = address of driver
A1 = address of the driver's side of the chip
D0.B = character to transmit

Exit: All registers should be the same as on entry.
The interrupt level must remain the same as on entry.

7.5.1.4 CK_TBE: Check to See if the Transmit Buffer Is Empty. CK_TBE tests to see if the transmit buffer is empty. This routine is called by XMIT in TERMLIB. Entry and exit conditions are described below.

Entry: A5 = address of CCB
A4 = address of the driver
A1 = address of the driver's side of the chip

Exit: All registers should be the same as on entry.
CCR: <NE> = transmit buffer empty
<EQ> = transmit buffer not empty
The interrupt level must remain the same as on entry.

7.5.1.5 DDP_RESET: Device-Dependent Reset. The DDP_RESET routine performs the device-dependent reset. The TERMLIB RESET routine clears the queues, flags, and other device-independent items. DDP_RESET is called by DO_HALT, DO_CONFIGURE, RECV, and BREAK in TERMLIB. Entry and exit conditions are described below.

Entry: A5 = address of CCB
A4 = address of driver
A1 = address of the driver's side of the chip
A0 = address of event in some cases

Exit: All registers should be preserved.
The interrupt level must remain the same as on entry.

7.5.1.6 SETUP: Set Up the Device According to the Configuration. SETUP uses the working configuration in the CCB to set up the device. This routine is called by DO_CONFIGURE. Entry and exit conditions are described below.

Entry: A5 = address of CCB
A4 = address of driver

Exit: All registers should be the same as on entry.
The interrupt level must remain the same as on entry.

7.5.1.7 CLOCK_RESET: Reset the Clock. CLOCK_RESET resets the onboard clock, if one exists, to its default value. If there is no clock, this routine returns (RTS). The CLOCK_RESET routine is called by DO_CONFIGURE. The device-dependent routine may use CLOCK_RESET while in DETECT_BAUD mode. Entry and exit conditions for CLOCK_RESET are described below.

Entry: A5 = address of CCB
A4 = address of driver
A1 = address of the driver's side of device
A0 = address of Configure/Status Block

Exit: All registers should be the same as on entry.
The interrupt level must be restored to what it was on entry.

7.5.1.8 GET_STAT: Get Device Status. GET_STAT prepares the device status byte which contains, if the module is capable of sensing it, the condition of the DSR line (which indicates device ready/unready). It sets such bits as XDSDCD, XDSABR, and XDSNRB in the MPSC driver. This routine does not set the break indicator in the device status byte.

GET_STAT is called by

a. REQ_STAT, a handler for the IOSTAT command. On return from GET_STAT, D0.B is moved into the IOSDST field of the user's IOCB.

b. CHNG_DEF, the handler for the IOCHDC command. On return from GET_STAT, D0.B is moved into the IOSDST field of the user's CSB.

c. IO_COMN, the handler for the READ, WRITE, and OUTWINP commands. On return from GET_STAT, the XDSNRB (not ready) bit is checked; if the device is not ready, the ISTANR (not ready) error message is returned.

d. BREAK, a background routine. If the driver receives a break while performing I/O, then it sends back an unsolicited event to IOS. On return from GET_STAT, the driver sets the XDSBRK (break) bit and returns the contents of D0 in the XRPDST (device status) field of the event.

e. DO_CONFIGURE, to put the latest status in the user's CSB.

Entry and exit conditions for GET_STAT are described below.

Entry: A5 = address of CCB
A4 = address of driver
A3 = address of IOCB (IO_COMN)
A1 = address of user's data block (REQ_STAT)
A1 = address of event (BREAK)
A1 = address of data to copy into DCB (CHNG_DEF)
A0 = address of user's CSB (CHNG_DEF and DO_CONFIGURE)
A0 = address of event area (BREAK)

Exit: All registers should be restored.
The interrupt level should be the same as on entry.

7.5.1.9 DDP_STOP: Device-Dependent STOP. DDP_STOP stops the device from transmitting. DDP_STOP is called by STOP, a background routine in TERMLIB. If the driver is not configured for XON/XOFF, then TERMLIB calls on DDP_STOP to physically act on the device to stop transmission. Entry and exit conditions are described below.

Entry: A5 = address of CCB
A4 = address of driver
A1 = address of the driver's side of device

Entry interrupt level: INHIBITED

Exit: All registers should be the same as on entry.
The interrupt level must remain the same as on entry.

7.5.1.10 DDP_UNSTOP: Device-Dependent UNSTOP. DDP_UNSTOP is the device-dependent part of the UNSTOP routine in TERMLIB. When the driver is not using XOFF/XON, it unstops the device in its own device-dependent way. When this routine is called, the driver is running in background mode, at interrupt level 0. DDP_UNSTOP is called by UNSTOP in TERMLIB.

Entry: A5 = address of CCB
A4 = address of driver
A1 = address of the driver's side of device

Entry interrupt level: INHIBITED

Exit: All registers should be preserved.
The interrupt level must remain the same as on entry.

7.5.1.11 DDP_BEG_BREAK: Device-Dependent Begin BREAK. DDP_BEG_BREAK writes whatever it must to the chip to transmit a break. DDP_BEG_BREAK is called by TM_BREAK and by BEG_BREAK, a background routine in TERMLIB. DDP_BEG_BREAK is called by TM_BREAK in TERMLIB. Entry and exit conditions are as follows.

Entry: A5 = address of CCB
A4 = address of driver
A1 = address of the driver's side of device

Entry interrupt level: INHIBITED

Exit: All registers should be preserved.

7.5.1.12 DDP_END_BREAK: Device-Dependent End BREAK. DDP_END_BREAK writes whatever it must to the chip to prevent transmission of a break. DDP_END_BREAK is called by STOP_BRK, a special interrupt service routine for the timer interrupt to stop the break signal, if the driver is in transparent mode. DDP_END_BREAK is also called by END_BREAK, a background routine in TERMLIB. Entry and exit conditions are described below.

Entry: A5 = address of CCB
A4 = address of driver
A1 = address of the driver's side of device

Entry interrupt level: INHIBITED

Exit: A5 = address of CCB
D0,A0,A1,A4 = irrelevant
All other registers must be preserved.

7.5.2 Required INCLUDE Files

The following INCLUDE files are normally required for most device drivers:

9995.&.IOE.EQ
9995.&.STR.EQ
9995.&.TCB.EQ
9995.&.NIO.EQ
9995.&.LV5.EQ
9995.&.CCB.EQ
9995.&.TERMINAL.EQ
9995.&.UTILITY.MC
9995.&.TERMCCB.EQ
9995.&.BAB.EQ

7.5.3 An Example: The MPSC Driver Structure

Figure 7-5 illustrates the structure of the MPSC driver, which can be used as a model for serial drivers under VERSAdos. Device-independent code was moved to TERMLIB, leaving only the device-dependent driver code in the MPSC driver.

```
|-------------------|
| SERVICE VECTOR    |
| TABLE             |
|                   |
|                   |
|                   |
|-------------------|
| BRANCH TABLE TO   |
| DEVICE-DEPENDENT  |
| ROUTINES          |
|                   |
|                   |
|-------------------|
| INIT:             |
|    BSR TERM_INIT  |
|    IF  <NE>  THEN |
|       IF <PL>  THEN
|           TRO$.KILLER ,
|       ELSE        |
|          RTS      |
|       ENDI        |
|    ENDI           |
|    LEA  DO_INIT(PC),A0
|    MOVE.L  #TOGUARD,D0
|    TRAP  #0       |
|    RTS            |
|                   |
|-------------------|
| COMMAND:          |
| Do any device-dependent tasks
| BSR.L TERM_COMMAND|
| RTS               |
|                   |
|-------------------|
| INTERRUPT:        |
| Jump table of entry
| points used by    |
| supervisor driver |
|                   |
|-------------------|
| RCA400:           |
| RCA:              |
|     .             |
|     .             |
|     .
|   BSR.L  TERM_GOT_CHAR
|   BRA.L  INT_EXIT
|                   |
|-------------------|
```

FIGURE 7-5. MPSC Driver Structure (Sheet 1 of 4)

```
| ES400:              |
| ES:                 |
|       .             |
|       .             |
|       .             |
|  If a DCD transition,
|    BKGRND  DCD      |
|       .             |
|       .             |
|       .             |
|  If a break,        |
|    BSR.L TERM_BREAK
|       .             |
|       .             |
|       .             |
|   BRA.L   INT_EXIT  |
|                     |
|---------------------|
| SRC400:             |
| SRC:                |
|       .             |
|       .             |
|       .             |
|   BSR.L   LOG_ERR   |
|       .             |
|       .             |
|   BRA.L   INT_EXIT  |
|                     |
|---------------------|
| DSR400:             |
|    BKGRND DSR       |
|    BRA.L   INT_EXIT |
|                     |
|                     |
|---------------------|
| RI400:              |
|       .             |
|       .             |
|   BKGRND DSR        |
|       .             |
|       .             |
|       .             |
|   BRA.L INT_EXIT    |
|                     |
|---------------------|
| INT_EXIT:           |
| Restore registers   |
| Set the carry bit   |
| and RTS             |
|                     |
|                     |
|---------------------|
```

FIGURE 7-5. MPSC Driver Structure (Sheet 2 of 4)

```
| CLOCK_RESET:        |
|                     |
|                     |
|                     |
|                     |
|---------------------|
| PUT_CHAR:           |
|                     |
|                     |
|                     |
|                     |
|---------------------|
| CK_TBE:             |
|                     |
|                     |
|                     |
|                     |
|---------------------|
| DDP_RESET:          |
|                     |
|                     |
|                     |
|                     |
|---------------------|
| SETUP:              |
|                     |
|                     |
|                     |
|                     |
|---------------------|
| GET_STAT:           |
|                     |
|                     |
|                     |
|                     |
|---------------------|
| DDP_STOP:           |
|                     |
|                     |
|                     |
|                     |
|---------------------|
| DDP_UNSTOP:         |
|                     |
|                     |
|                     |
|                     |
|---------------------|
```

FIGURE 7-5. MPSC Driver Structure (Sheet 3 of 4)

```
| DDP_BEG_BREAK:    |
|                   |
|                   |
|                   |
|                   |
|-------------------|
| DDP_END_BREAK:    |
|                   |
|                   |
|                   |
|                   |
|-------------------|
| DO_INIT:          |
|                   |
| Do the required setup for TERMLIB
| Do any device-dependent tasks
| BSR.L  RESET      |
| BSR.L  DDP_RESET  |
|                   |
| If channel is down,
| BSR.L MARK_DOWN   |
|                   |
|   RTS             |
|-------------------|
| DSR0:             |
| DSR:              |
|                   |
|  If driver is waiting for DSR high,
|   BKGRND  BREAK   |
|  ELSE             |
|   BSR.L TERM_UNRDY|
|   IF  <NE>  THEN  |
|     TR0$.KILLER , |
|   ENDI            |
|                   |
|  RTS              |
|-------------------|
| DCD0:             |
| DO_DCD:           |
|                   |
| If driver is to report DCD transitions,
|  BKGRND  BREAK    |
|                   |
| If driver is configured to do auto baud rate detection,
|  BSR.L RESET      |
|  BSR.L DDP_RESET  |
|  BSR.L CLOCK_RESET|
|       .           |
|       .           |
|                   |
|   RTS             |
|-------------------|
```

FIGURE 7-5. MPSC Driver Structure (Sheet 4 of 4)

7.6 SYSGENING THE NEW DRIVER INTO VERSAdos

7.6.1 TCHTYPE.AG

The TCHTYPE.AG file is assembled at SYSGEN time, and a listing is included in the SYSASML.LS listing. This file contains a table of information for each serial port that may use TERMLIB. TCHTYPE.RO is linked with TERMLIB.RO to produce TERMLIB.TF at SYSGEN time. Figure 7-6 illustrates the TERMLIB.TF file structure.

```
           +-------------------+
+-----------  link to TCHTYPE  |
|          |                   |
|          |  TERMLIB.RO       |
|          |                   |
|          |                   |
|          |-------------------|
|          |                   |
+----------->  TCHTYPE.RO      |
           |                   |
           +-------------------+
```

FIGURE 7-6. TERMLIB.TF Structure

Each port that wants to use TERMLIB must have an entry in TCHTYPE with the following information:

Channel type (byte)
Driver code (byte)
Attributes mask (word)
Parameters mask (word)
Mask of recognized baud rates (longword)

This required information is described in the following paragraphs.

7.6.1.1 Channel Type. The channel type is defined in file 9995..IOE.EQ, and is a number in the range $60 through $7F. Some of the channel types that have already been defined are as follows:

XTS7A2	$60	VM02, Side A
XTS7B2	$61	VM02, Side B
XTS7AR	$64	MVME400, Side A
XTS7BR	$65	MVME400, Side B
	$69	Reserved
XTSMFP	$6C	MVME120 (only one side)
	$6D	Reserved
XTMPC1	$72	MVME050, Port 1
XTMPC2	$73	MVME050, Port 2
XTS7A3	$74	VM03, Side A
XTS7B3	$75	VM03, Side B
XTSIO0	$76	VM04, Port 0
XTSIO1	$77	VM04, Port 1
XTSCC0	$78	MVME117, Port 0
XTSCC1	$79	MVME117, Port 1

If a module has two ports, then port A is assigned the even number and port B the next odd number. If a module has only one port, then its port is assigned an even number and the next odd number remains unassigned.

In the TERM_INIT routine in TERMLIB, the TCHTYPE table is searched, and the channel type for each entry is compared with the channel type from the CCB.

If the channel type is found in the TCHTYPE table, the information that follows is copied into the CCB in the DRV_CODE, REC_ATT, REC_PAR, and REC_BAUD fields. TERMLIB divides the channel type by 2 to determine the value for the BOARD entry in the CCB. The BOARD entry is used by the MPSC driver to determine if the device it is currently driving is an M68KVM02 or an MVME400 module.

If the channel type is not found, then the channel is marked down.

7.6.1.2 Driver Code. The driver code is defined in file 9995..IOE.EQ. Some of the driver codes that have already been defined are as follows:

IODMPSC	$01	MPSC driver for VM02, VM03, and MVME400
IODMPCC	$03	Terminal driver for MVME050
IODMFP	$0B	MFP driver for MVME120
IODSIO	$0E	Terminal driver for VM04
IODVM22	$21	Driver for VM22 disk controller

7.6.1.3 Attributes Mask. The attributes mask is the mask of all attributes that are recognized by the driver. Any Configure or Configure-Defaults command must have an attributes mask in its IOCB that is a subset of the attributes mask found in TCHTYPE.

7.6.1.4 Parameters Mask. The parameters mask is the mask of all parameters that are recognized by the driver. Any Configure or Configure-Defaults command must have a parameters mask in its IOCB that is a subset of the parameters mask found in TCHTYPE.

7.6.1.5 Mask of Recognized Baud Rates. The baud rate codes are consecutive numbers in the range $00 through $1F. Each of these codes corresponds to a baud rate (refer to the VERSAdos Data Management Services and Program Loader User's Manual). Setting a bit in the longword in this entry in TCHTYPE means that the corresponding baud rate code is legal for this channel. Clearing the bit means that the corresponding baud rate code is not legal for the channel.

7.6.2 MYDRIVER.CI

The following file is INCLUDEd at SYSGEN time to add the new driver.

In <system>.CNFGDRVR.CI:

```
        .
        .
        NOMYBOARD = 1
        .
        .
        .
```

In <system>.IFDRVR.CI:

```
        IFGT    \NOMYBOARD
        INCLUDE   MYDRIVER.CI
        ENDC
        .
        .
        .
```

MYDRIVER.CI:

```
  MYDRIVER = *
  SUBS          &.MYDRIVER.LG
  EXCLUDE       DRVL
  EXCLUDE       TERM
  LINK          &.MYDRIVER.LG
  IFEQ          \LINKLS
        =COPY         \LINKLS,\WORKLS;A
  ENDC
  PROCESS       &.MYDRIVER.LO
  END           &.MYDRIVER
```

7.6.3 MYDRIVER.LG

The following file is MYDRIVER.LG:

```
=/*
=/*  MYDRIVER.LG -- SYSGEN chainfile to link new subordinate driver.
=/*
=/*  SYSGEN parameter LINKLS = \LINKLS = file/device to which
=/*  to send the linker listing.
=/*
=/*  SYSGEN parameter MYDRIVER = \MYDRIVER =  address  at  which  to  link
=/*  driver.
=/*
=LINK ,&.MYDRIVER.LO,\LINKLS;HAMIXS
SEGMENT MYDR:0-13 \MYDRIVER
SEGMENT DRVL:15  \DRVL
SEGMENT TERM:14  \TERMLIB
INPUT &.MYDRIVER.RO
INPUT &.DRVLIB.RO
INPUT &.TERMLIB.TF
END
=END
```

CHAPTER 8

DEBUGGING THE DRIVER

8.1 INTRODUCTION

This chapter describes techniques for debugging device drivers which may be helpful in analyzing a repeating system crash or determining why a system does not boot.

8.2 DRIVER DEBUGGING TECHNIQUES FOR VERSAdos

When a system will not boot, the reason can often be traced to the addition of a new piece of code, in the form of a device driver or operating system tas' that would run during boot time. This paragraph explains suggested approache for debugging device drivers. It is not intended to restrict driver writer to one method of debugging.

EXORmacs Example:

When enough of the driver has been coded, it should be SYSGENed into the operating system and booted from MACSbug using the BH command. Breakpoints can then be set, using MACSbug, at points of interest in the driver, the addresses of which can be obtained from the SYSGEN map. Good breakpoints might be the start of the initialization, command service, and interrupt service routines. On the break, MACSbug can then be used to trace, display, or set memory in the normal way. (Refer to the note in the next example).

By repeatedly adding new code sections to the driver, SYSGENing in the driver, and booting the system with the MACSbug BH command, the new driver can be brought into service.

NOTE

> If the driver is SYSGENed in after other processes, the restart option of SYSGEN could be used to reduce SYSGEN time.

MVME120 Driver Example:

This example assumes that a device driver has been written for an MVME120 module and is being integrated into the VERSAdos operating system. At this point in the example, a SYSGEN has successfully been completed to include the code module.

Throughout this example, user entries are shown in **boldface type**.

Remembering that there are only three entry points to the driver, start by setting breakpoints at each of the entry points and then tracing from that point. To accomplish this end, first use the debug monitor (in this case, the MVME120 Debug Monitor) Boot Halt command.

`MVME120 x.y  > BO 0,0,;H`	Causes the Initial Program Loader (IPL) to bring in VERSADOS.SY and then return control to the MVME120 Debug Monitor.

Next, modify the illegal-instruction vector so that it remains pointing at the MVME120 Debug Monitor's breakpoint routine. Normally, when the processor is reset, the MVME120 Debug Monitor initializes all the MC68010 vectors from $0 to $400. When the operating system gains control at boot time, the vectors are initialized to point to the various system routines to handle the interrupt or exception.

The vector initialization table can be changed so that the system initialized does not change the illegal-instruction vector, which is at address $10. (Normally, the vector table is in the first $100 bytes of RMS68K). To do so, locate the string "!VCT" at offset 1180 in the memory dump below. This is the start of the Vector Control Table, each entry of which is as follows:

1 byte for vector #
3 bytes for address

For example, the following data is found at 1184:

0000 145E	The first byte (00) indicates vector 0. The next 3 bytes (00145E) hold the address that will be placed in vector 0.

The vector of interest in this example is vector #4. At address 1190 resides the information "0400 194E". This must be changed to "0400 0001". By putting a 1 in the address field, the system initializer is instructed to skip this vector, which effectively leaves it pointing to the MVME120 Debug Monitor.

```
=SYSANAL
SYSANAL VERSION 021585 4
SYSA: >MD 1100 100
001100  001100  6100  01F4  6100  3E2A  6000  04BA  4EB9  0000  a...a.>*'...N...
001110  001110  1FB6  4455  4D59  2278  0D2C  007C  0700  2469  ..DUMY"x.,....$i
001120  001120  0004  B5FC  0000  0000  6606  6100  0E8A  604A  ........f.a...'J
001130  001130  2352  0004  027C  F8FF  4292  42AA  0004  257C  #R......B.B...%.
001140  001140  7FFF  FFFF  0008  42AA  000C  47FA  FFCA  254B  ......B...G...%K
001150  001150  0010  426A  0014  257C  FFFF  FFFE  0016  357C  ..Bj..%.......5.
001160  001160  F8FF  001C  007C  0700  2669  0008  0C93  0000  ........&i......
001170  001170  0000  6704  47D3  60F4  268A  027C  F8FF  4E73  ..g.G.'.&.....Ns
001180  001180  2156  4354  0000  145E  0200  194A  0300  194C  !VCT...^...J...L
001190  001190  0400  194E  0500  1950  0900  1960  0A00  1958  ...N...P...'...X
0011A0  0011A0  0C00  0000  0E00  1FB6  0F00  1FB6  1000  0000  ................
0011B0  0011B0  1800  2418  1900  123C  1A00  123C  1B00  123C  ..$....<...<...<
0011C0  0011C0  1C00  123C  1D00  123C  1E00  123C  1F00  0001  ...<...<...<....
0011D0  0011D0  2000  2516  2100  25B4  2200  18DE  3000  0000   .%.!.%."...0...
0011E0  0011E0  5000  0000  5100  0000  5200  0000  5300  0000  P...Q...R...S...
0011F0  0011F0   100  0053  6000  145E  6100  145E  6200  145E  ...S'..^a..^b..^
```

NOTE

This procedure is effective only for debugging operating system tasks and renders normal SYMbug breakpoints ineffective. (Refer to the SYMbug/A and DEbug Monitors Reference Manual.)

After patching to allow MVME120 Debug Monitor breakpoints, determine where to set the first breakpoint. Using the listing obtained from the SYSGEN, locate the start of the driver under consideration. The first three longwords will be offsets to the various entry points. Add the start of the driver address to each of the offsets and set breakpoints at each address. The simple way of doing this is to display the memory at the start of the driver using the Memory Display (MD) command. This procedure is illustrated as follows:

```
MVME120 x.y > MD 7700 20
007700      00 00 09 CE 00 00 00 FA  00 00 00 28 00 00 00 00  ..N...z...(....
007710      02 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ...............

MVME120 x.y > BR 7700+9CE 7700+FA 7700+28

Breakpoints
0080CE  0080CE
0077FA  0077FA
007728  007728

MVME120 x.y >
```

Set the program counter; the startup address is also in the MD listing.

```
   MVME120 x.y > .PC
   MVME120 x.y > G
```

The system will begin executing, and will continue until one of the breakpoints is encountered. (Normally, the first breakpoint would be the INIT entry point in the device driver.) At this point, control of the processor resides with the user; it is available for tracing, setting other breakpoints, displaying the code, or simply continuing until the next breakpoint.

8.3 I/O EVENT STRUCTURE

When a driver completes an I/O operation for a channel, the driver must notify the task that is attached to the channel by queuing an I/O completion event to the task in question. Under normal conditions the task attached to the channel is IOS, the I/O Services module.

Sometimes an I/O completion event is queued that is not properly formatted, and this can result in errors. When debugging driver code, it is important to ensure that all I/O events returned by the driver are formatted correctly.

I/O completion event queuing is made necessary by the following circumstances. Input and output transactions are performed asynchronously under interrupt control, so execution of the task that originally makes a request to initiate an I/O transaction is not inherently synchronized with completion of the I/O transaction. When the driver receives a request to initiate an I/O transaction, the driver sets up variables indicating that an I/O transaction should be performed under interrupt control. Then, without waiting for any I/O operation to be completed, the driver returns control to the routine (normally the CMR handler) that called the driver. The CMR handler then returns control to the task (normally IOS) that invoked the CMR handler to initiate the I/O transaction. As driver interrupts occur, the CMR handler answers the interrupts and invokes the driver's interrupt service routine to handle the interrupts. Eventually, the driver completes the requested I/O transaction under interrupt control; it then queues an I/O completion event to tell the attached task (usually IOS) that the I/O transaction has been completed.

In practice, the driver actually interacts with the CMR handler and IOS as described above, but nothing in the driver restricts it to interactions with those particular routines. Everything is done on a general basis, so the driver can be used as it is written in new and unanticipated environments.

At times, an I/O driver must queue an unsolicited event to the task that is attached to its channel. An unsolicited event is an event that occurs spontaneously, rather than as a result of a task's request for an I/O transaction. For example, an I/O driver might queue an unsolicited event to tell the task attached that an I/O device has been taken offline. Similarly, an I/O driver might queue an unsolicited event to tell the attached task that an I/O device has been put back online. These are unsolicited events, occurring independently rather than in response to actions of the task attached to the channel.

An event is defined by system software as a sequence of bytes to be transferred to a task. The bytes in an event contain information meaningful to the task that receives the event. An I/O event is only one of many different kinds of events. The structure of an I/O event can be diagrammed as follows.

```
     |----------------|
$00  | EVNTLEN        |  <-- (Length of the event in terms of bytes)
     |----------------|
$01  | EVNTCODE       |  <-- (Code = $01 or $81 for an I/O event)
     |----------------+----------------+----------------+----------------|
$02  : EVNTSVAD (Logical event service address; present if code = $81)  :
     :----------------+----------------+----------------+----------------:
$02  | EVNTTYPE       |  <-- (Type:  completion = $70; unsolicited = $80)
     |----------------|
$03  | EVNTKEY        |  <-- (Used by attached task to identify channel)
     |----------------+----------------+----------------+----------------|
$04  | EVNTDCB (Physical DCB address) or EVNTCID (Channel ID mnemonic)   |
     |----------------+----------------+----------------+----------------|
$08  | EVNTSTAT (Status value)         |
     |----------------+----------------|
$0A  : EVNTDST (Device status)         :  <-- (Present if EVNTTYPE = $80)
     :----------------+----------------:
$0C  : EVNTDNUM       :  <-- (Device number; this field and the reserved
     :----------------:       field below it are present only if
$0D  : Reserved       :       EVNTTYPE = $80 and EVNTSTAT = $01)
     :----------------:
```

Note 1. The EVNTSVAD field is optional. It is present in an I/O event only if EVNTCODE = $81. If the EVNTSVAD field is present, it specifies the logical address of the routine that should be invoked to respond to the event. Events are normally handled by a task's ASR, but a task can specify a special routine to be invoked in response to an I/O event from a particular channel. If the EVNTSVAD field is present, the offset values given for the fields following the EVNTSVAD field must be biased by an increase of 4.

Note 2: If the EVNTSVAD field is present in an event, RMS68K processes the EVNTSVAD field and removes it from the event before the event is given to the destination task. Therefore, the receiving task is not concerned with the EVNTSVAD field. Only the sending task and the RMS68K Executive must deal with the optional EVNTSVAD field.

Note 3: The EVNTDCB field is normally used to specify the physical DCB address for the device that is associated with the I/O event. If no DCB address is available, the EVNTDCB field is replaced with an EVNTCID field, which contains the channel ID mnemonic of the channel that is associated with the event. This situation arises when an unsolicited device event occurs before a DCB address has been specified. In this case, EVNTTYPE = $80 indicates an unsolicited device event, and EVNTSTAT = $01 indicates the DCB address is not available for the unsolicited device event.

Note 4: The EVNTDST (device status) field is present only for unsolicited events, so the EVNTDST field is present in an event only if EVNTTYPE = $80.

Note 5: The EVNTDNUM (device number) field and the reserved field below it are present only if EVNTTYPE = $80 and EVNTSTAT = $01. This combination indicates an unsolicited device event with no DCB address given. Because a channel ID mnemonic is supplied in place of a DCB address, the EVNTDNUM field is included to tell the attached task which drive is associated with the event. When the DCB address is available, it can be used to determine which drive is involved.

The individual fields of an I/O event are defined by the equates that follow, and the fields are described in detail as they are defined.

	OFFSET	0	This OFFSET block defines the structure of an I/O event. One variation of the event structure indicates an I/O completion event; another variation indicates an unsolicited device event.
L0010	EQU	*	This label marks the first byte of an event block. It serves as a base address for computing the lengths of certain kinds of events.
EVNTLEN	DS.B	1	This field specifies the length of the event. The length is specified in terms of the number of bytes in the event.
EVNTCODE EVNTIO EVNTIOSA	DS.B EQU EQU	1 $01 EVNTIO+$80	This field contains the event code, which specifies the classification of the event. The event code for an I/O event can be $01 or $81. The $81 event code indicates the I/O event includes a logical event service address (refer to next paragraph), and the $01 event code indicates the I/O event does not include a logical event service address.
EVNTSVAD	DS.L	1	This field is present in an I/O event only when EVNTCODE = $81. If the EVNTSVAD field is present it contains the 4-byte logical address of the routine to be invoked in response to the I/O event. The logical address is an address in the task (normally IOS) that is attached to the channel. Events are normally handled by a task's ASR. However, when a task attaches itself to a channel, the task can designate a special routine to be invoked in response to an I/O event from that particular channel.

If the attaching task designates a special event-handling routine at attach time, the CMR handler puts the logical address of that event-handling routine into the CCBSVVC field of the channel's CCB. If no routine is specified, the CMR handler clears the CCBSVVC field. The device driver uses the CCBSVVC field of the CCB to determine the value (if any) for the EVNTSVAD field of an I/O event.

EVNTSVSZ	EQU	*-EVNTSVAD	If the optional EVNTSVAD field is present in an event, the values of the labels for all the following event fields must be biased by an increase of EVNTSVSZ to account for the presence of this optional 4-byte field.
	OFFSET	EVNTSVAD	Resets the location counter to define the remaining event fields as if the optional EVNTSVAD field were not present.
EVNTTYPE	DS.B	1	This field specifies the type of I/O event that is being queued. The following values are defined for this field.
EVNTCOMP	EQU	$70	EVNTTYPE = $70 indicates an I/O completion event. An I/O completion event is normally queued to signify the completion of a requested I/O transaction.
EVNTHALT	EQU	$71	EVNTTYPE = $71 indicates a HALT/ABORT event.
EVNTUNSD	EQU	$80	EVNTTYPE = $80 indicates an unsolicited device event.
EVNTUNSC	EQU	$FF	EVNTTYPE = $FF indicates an unsolicited channel event.
EVNTKEY	DS.B	1	The value of the event key field comes from the CCBKEY field of the CCB, which was defined earlier. This key value is specified at ATTACH time by the task that attaches itself to the channel, and the key can be used (or ignored) by that task as it chooses. The attached task might profitably use the event key as a shorthand channel identifier to determine which one of several attached channels is the source of an event.
EVNTDCB	DS.L	1	This field normally contains the physical address of the DCB for the device that is associated with the I/O event. Therefore, this field tells the attached task which device is associated with the I/O event.

EVNTCID	EQU	EVNTDCB	In some cases an unsolicited I/O device event can occur before any task has initiated any I/O transactions with the I/O device in question. In such cases no DCB address is available, so a channel ID mnemonic is supplied in place of the EVNTDC8 field that is normally part of an I/O event. The task receiving the event recognizes that a channel ID mnemonic replaces the DCB address because EVNTTYPE = $80 indicates an unsolicited device event and EVNTSTAT = $01 indicates that the DCB address is not available for the unsolicited device event. When the DCB address is not available, an extra field is provided (refer to EVNTDNUM description) in the event to tell the receiving task which device on the channel is associated with the event.
EVNTSTAT	DS.W	1	For an I/O completion event, the event status field contains the same value that the I/O driver returns in the IOSSTA field of the IOCB. For an unsolicited device event (which is identified by EVNTTYPE = $80), the EVNTSTAT field is used differently. In this case, EVNTSTAT = $00 if the DCB address is supplied, and EVNTSTAT = $01 if the DCB address is not supplied.
EVNTCMSZ	EQU	*-L0010	This label defines the size (in bytes) of a normal I/O completion event.
EVNTDST	DS.W	1	The device status field is not part of an I/O completion event, but is included in an unsolicited I/O device event. The value that appears in the device status field is analogous to the value that is returned in the IOSDST field of a configure/status parameter block. Because the device status value occupies only one byte, the most significant byte of this field is always zero.
EVNTUNSZ	EQU	*-L0010	This label defines the size (in bytes) of an ordinary unsolicited I/O device event.
EVNTDNUM	DS.B	1	The EVNTDNUM (device #) field and the reserved field below it are present only if EVNTTYPE = $80 and EVNTSTAT = $01. This combination indicates that an unsolicited device event has occurred with no DCB address available. Because a channel ID mnemonic is supplied in place of a DCB address, the EVNTDNUM field is included to inform
	DS.B	1	

		the attached task which drive is associated with the event.
EVNTDNSZ EQU	*-L0010	This label defines the size (in bytes) of an unsolicited I/O device event which includes the EVNTDNUM field.
EVNTMXSZ EQU	*-L0010 +EVNTSVSZ	This label defines the maximum possible size (in bytes) of an event that has all fields, including the optional EVNTSVAD field.

THIS PAGE INTENTIONALLY LEFT BLANK.

APPENDIX A

THE CHANNEL MANAGEMENT REQUEST (CMR) HANDLER

A.1 INTRODUCTION

This appendix describes the functions available through the CMR handler and the error codes produced. Information for invoking the CMR handler is also included.

A.2 CMR HANDLER

User task interactions with channels are performed through requests made to the CMR handler. The available CMR functions are listed in paragraph 2.3.1.

Normally, a channel would be allocated when the system is initialized. When a user task wished to perform an I/O function on a particular device, it would attach to the appropriate channel. The user task would then be able to initiate I/O. When the I/O function is complete, the user task could detach from the channel or remain attached to the channel for future I/O requests.

A.2.1 Invoking the CMR Handler

A user task makes requests to the CMR handler through an RMS68K directive. A user task issues this directive to RMS68K by executing a TRAP #1 instruction (see Figure 2-3). Register D0 must contain the directive number of value 60, and register A0 must contain a pointer to a parameter block. The parameter block format varies according to the exact request. The format for each particular request can be found in the following paragraph.

An example of a user task making a CMR handler request is shown here:

```
UTSK:   .
        .
        .
        MOVE.L   60,D0        LOAD DIRECTIVE NUMBER
        LEA      PRMBLK,A0    LOAD PARAMETER BLOCK POINTER
        TRAP     1            ISSUE DIRECTIVE
        .
        .
        .
PRMBLK:                       DEFINE THE PARAMETER BLOCK
```

RMS68K will invoke the CMR handler when a directive number 60 is issued. The CMR handler then interprets the contents of the parameter block to determine the actions to be taken. When the CMR handler receives a request to initiate I/O, it will invoke one of the I/O handlers. The particular I/O handler invoked depends upon the service address supplied when the channel was allocated. There is one I/O handler for each channel.

A.2.2 Return from the CMR Handler

When a user task issues a request to the CMR handler, the task is placed in the READY state list. When that task regains control, the CMR handler will have acted upon the request. The low order byte of register D0 will contain zero if the request was successfully completed, or it will contain an error code if the request was not successfully completed. The Z bit of the status register will reflect the contents of register D0: Z = 1 if register D0 = 0; otherwise Z = 0. The value of the Z bit can be tested by using the MC68000 Bcc instruction (BEQ, BNE).

It is important to realize that if an INITIATE request is made to the CMR handler, the error code returned by the CMR handler indicates merely the successful or unsuccessful invocation of the I/O handler. It does not reflect the successful or unsuccessful completion of the actual I/O function performed by the I/O handler.

Refer to the paragraph entitled "Error Codes" in this chapter for a complete description of error codes.

A.2.3 ALLOCATE - Allocate a Channel

Parameter Block Format:

Field #		
1	1 byte	Code = $01
2	1 byte	Not used
3	2 bytes	Options
4	4 bytes	Channel Mnemonic
5	1 byte	Channel Type
6	1 byte	Reserved (used by CHPI)
7	4 bytes	I/O Handler Service Address
8	4 bytes	Supervisor Channel Mnemonic (used only if option bit 3 set)
9	4 bytes	Base Address of Memory Mapped I/O Space
10	2 bytes	Length of Memory Mapped I/O Space
11	1 byte	Hardware Vector Number
12	1 byte	Hardware Priority Level
13	1 byte	Software Priority Level
14	1 byte	Number of Polling Table Entries
- Variable Length Polling Table Follows -		
15	2 bytes	Polling Byte Offset
16	1 byte	Polling Mask
17	1 byte	Polling Test Value
18	2 bytes	Reset Byte Offset
19	1 byte	Reset Value
20	1 byte	Reserved

Parameter Block Field Descriptions:

Options -

Bit 0 = 0	Attach requests from any task are honored.
= 1	Attach requests from system tasks only are honored.
Bit 1	Reserved.
Bit 2	Exclusive vectoring.
Bit 3	Channel is subordinate to Supervisor Channel Mnemonic field.
Bit 4	Channel is to be a supervisor.

Channel Mnemonic -

Uniquely identifies the channel. It must be nonzero and it must be distinct from the mnemonic for any other channel or volume name in the system.

Channel Type -

Code which indicates the physical type of the channel. Interpretation of this code will cause CMR software to handle the INITIATE I/O command in one of four ways. Channels with code $01 through $0F will pass the request to the I/O handler with no parameter block checking. Channels with code $10 through $8F will do parameter block checks. Channels with code $FF will not do parameter block checks; they are used only to notify a task that an interrupt occurred.

I/O Handler Service Address -

This field points to an I/O handler. The structure of the I/O handler must have a service vector table that defines:

- The interrupt service entry point.
- The INITIATE I/O command service entry point.
- The initialization entry point.
- A longword for future use.

These are all 4-byte fields of absolute addresses. This field is not required for channel type $FF, which assumes the interrupt is cleared when the CMR handler polls the device and finds it activated.

Base Address of Memory Mapped I/O Space -

Physical address of the lowest byte in memory of the memory mapped I/O space for this channel.

Length of Memory Mapped I/O Space -

Zero relative consecutive byte count of the memory mapped I/O space for this channel. All future references to the memory mapped I/O space for this channel made through the CMR handler must be within the boundaries defined by the base address for memory mapped I/O and this field.

Hardware Vector Number -

Indicates the associated hardware vector. It must be an auto vector (values 25-31) or a user vector (64-255).

Hardware Priority Level -

Indicates the hardware interrupt level associated with the channel. It must be a value in the range of 1-7, inclusive.

Software Priority Number -

Indicates the position of the Channel Control Block (CCB) within the polling chain. A higher value of the software priority level will result in faster service to the channel when it interrupts.

Number of Polling Table Entries -

Indicates the number of 8-byte polling table entries which follow. This number can be in the range of 1-4, inclusive. For each type of interrupt associated with a channel, one 8-byte table entry is required to describe the details of that interrupt. When an interrupt occurs, the following algorithm is used to determine if this particular channel caused an interrupt.

The polling byte defined by the polling byte offset is read. If the polling test value is zero, the polling byte is complemented; otherwise, it is left unchanged. The resulting polling byte is then ANDed with the polling mask. If this result is nonzero, it is assumed that this channel caused the interrupt.

These entries are used only for interrupt-only and nonstandard channels.

Polling Byte Offset -

The zero-relative offset from the base of memory mapped I/O space for this channel where the polling byte resides. Refer to the polling algorithm described in "Number of Polling Table Entries".

Polling Mask -

Used in the polling algorithm to determine if this channel caused an interrupt. Refer to the polling algorithm described in "Number of Polling Table Entries".

Polling Test Value -

Used in the polling algorithm to determine the polling byte value. Refer to the polling algorithm described in "Number of Polling Table Entries".

Reset Byte Offset -

The zero-relative offset from the base of memory mapped I/O space for this channel where the reset byte resides.

Reset Value -

Used by the interrupt service to clear random or unexpected interrupts on a channel. It is critical that the reset byte offset and reset value be defined correctly to prevent infinite loops in the polling routine caused by interrupts that cannot be cleared.

Request Function Description:

A CCB is allocated and initialized for the channel with the specified mnemonic. The CCB is linked into the interrupt polling chain according to the specified software priority number.

Channels must be allocated by system tasks. After a channel has been allocated, other tasks may attach to that channel and initiate I/O.

After a channel is established, the CMR handler will turn control over to the service routine at its initialization entry point with a JSR instruction. This allows a driver, for example, to initialize a device to a known state before the operating system is running.

A.2.4 ATTACH - Attach a Channel

Parameter Block Format:

Field #		
1	1 byte	Code = $03
2	1 byte	Not used
3	2 bytes	Options
4	4 bytes	Channel Mnemonic
5	1 byte	User Generated ID
6	1 byte	Length of ASQ Return Entry
7	4 bytes	ASQ Service Vector
8	2 bytes	Reserved
9	2 bytes	Reserved

Parameter Block Field Descriptions:

Options -

Bit 0 = 0	Issue a WAKEUP directive for requesting task when I/O is complete.
= 1	Return I/O completion event in requesting task's ASQ .

Return events for I/O completions on standard channels must be returned through the ASQ.

Channel Mnemonic -

A unique, nonzero identifier assigned to the channel when it was allocated.

User Generated ID -

The field can contain any value. This value is returned as part of the return packet upon I/O completion. It can be used by the requesting task to identify I/O completions when concurrent multiple I/O requests are being processed.

Length of ASQ Return Entry -

Applicable only when flag bit 0 = 1. This field specifies the maximum byte length of the event. If the value in this field is "n", the entire event, the first "n" bytes, or the first 32 bytes will be returned, whichever is shorter.

ASQ Service Vector -

Applicable only when flag bit 0 = 1. If this field is zero, the default ASQ service vector associated with the requesting task will be used for processing the I/O completion event. Otherwise, this field will specify the alternate ASR service vector which is to be used for processing the I/O completion event.

Request Function Description:

The channel is logically connected to the requesting task, which is then considered the channel driver. The channel is dedicated to the channel driver until the driver issues a DETACH request for the channel. This ATTACH request must be made before any INITIATE I/O requests are made.

NOTE

When using ATTACH under a VERSAdos operating system environment, users cannot ATTACH to a device already SYSGENed into VERSAdos. If VERSAdos is unaware of a device, it can be allocated and attached.

A.2.5 DELETE - Delete a Channel

Parameter Block Format:

Field #

1	1 byte	Code = $024
2	1 byte	Not Used
3	2 bytes	Not Used
4	4 bytes	Channel Mnemonic

Parameter Block Field Descriptions:

Channel Mnemonic -

A unique, nonzero identifier which is assigned to the channel when it is allocated. The channel must currently be attached to the requesting task.

Request Function Description:

The specified channel is detached from the requesting task, and the CCB of the specified channel is removed from the system.

A.2.6 DETACH - Detach a Channel

Parameter Block Format:

Field #		
1	1 byte	Code = $04
2	1 byte	Not Used
3	2 bytes	Options
4	4 bytes	Channel Mnemonic

Parameter Block Field Descriptions:

Flag -

Bit 0 = 0	Detach only the specified channel.
= 1	Detach all channels which are attached to the requesting task. A valid channel mnemonic must still be supplied.

Channel Mnemonic -

A unique, nonzero identifier of a channel which was assigned to the channel when it was allocated. This channel must be currently attached to the requesting task.

Request Function Description:

The logical connection between the specified channel and the requesting task is removed.

A.2.7 PUT ON LINE - Put a Channel Online

Parameter Block Format:

Field #		
1	1 byte	Code = $05
2	1 byte	Not Used
3	2 bytes	Not Used
4	4 bytes	Channel Mnemonic

Parameter Block Field Descriptions:

Channel Mnemonic -

The unique, nonzero identifier of the channel which is being put online. This mnemonic was assigned when the channel was allocated.

Request Function Description:

The specified channel is removed from offline status, allowing ATTACH requests for that channel to be honored.

A.2.8 TAKE OFFLINE - Take a Channel Offline

Parameter Block Format:

Field #		
1	1 byte	Code = $06
2	1 byte	Not Used
3	2 bytes	Not Used
4	4 bytes	Channel Mnemonic

Parameter Block Field Descriptions:

Channel Mnemonic -

A unique, nonzero identifier of the channel being taken offline. The channel must currently be attached to the requesting task. The channel mnemonic was assigned when the channel was allocated.

Request Function Description:

The specified channel is given an offline status which prevents any ATTACH requests from being performed on this channel. The channel is detached from the requesting task. The channel's definition remains in the system. The channel remains in offline status until a PUT ON LINE request is made.

A.2.9 INITIATE - Initiate I/O on a Channel

Parameter Block Format:

The first six fields are identical for initiating I/O on both standard and nonstandard channels. After the first six fields, the parameter block fields may differ.

- Standard and Nonstandard Channel Common Fields -

Field #		
1	1 byte	Code = $07
2	1 byte	Subcode = 0
3	2 bytes	Not Used
4	4 bytes	Channel Mnemonic
5	4 bytes	Taskname of Task Containing Buffer Space
6	4 bytes	Session Number of Task Containing Buffer Space

- Standard Channel Unique Fields and Shared-Access -

7	4 bytes	User-Supplied (DCB Address for Standard Channels)
8	4 bytes	Logical Base Address of Command Packet (IOCB)
9	2 bytes	Packet Length

Parameter Block Field Descriptions:

Channel Mnemonic -

The unique, nonzero identifier of the channel on which I/O is being initiated. This mnemonic was assigned when the channel was allocated.

Taskname of Task Containing Buffer Space -

Name of the task in which the buffer space resides. If this field is zero, the session number of the requesting task is assumed. If the requesting task is not a system task, this session number must equal the session number of the requesting task.

User-Supplied ID -

This ID is passed back as part of the completion event.

Logical Address of Command Space -

This command space contains the command packet which the I/O handler will use. The command packet must reside within the address space of the requesting task.

Packet Length -

Length of the packet supplied.

Request Function Description:

The appropriate I/O handler is invoked. For standard and shared-access channels, the taskname information and parameter block structure are checked. For nonstandard channels, no checking is done. For shared-access channels, return events are always through the ASQ.

A.2.10 HALT - Halt I/O

Parameter Block Format:

Field #		
1	1 byte	Code = $07
2	1 byte	Subcode = $02
3	2 bytes	Not Used
4	4 bytes	Channel Mnemonic

Parameter Block Field Descriptions:

Channel Mnemonic -

The unique, nonzero identifier of the channel on which I/O is being halted. The mnemonic was assigned when the channel was allocated.

Request Function Description:

This request is applicable for nonstandard channels only. This request is passed directly to the I/O handler.

A.2.11 RESET - Reset Interrupt

Parameter Block Format:

Field #		
1	1 byte	Code = $07
2	1 byte	Subcode = $07
3	2 bytes	Not Used
4	4 bytes	Channel Mnemonic

Parameter Block Field Descriptions:

Channel Mnemonic -

The unique, nonzero identifier of the channel for which a reset of hardware interrupts is desired. The mnemonic was assigned when the channel was allocated.

Request Function Description:

Applicable to standard and shared-access channels only. This request is passed directly to the I/O handler.

A.2.12 Error Codes

Error codes are returned to a driver task from the CMR handler. Error codes are classified into three types, as follows:

Type	Classification
00	General system-detected error. These errors are common to those produced by all RMS68K directives. The CMR handler is capable of returning a subset of the entire set of general system-detected error codes.
10	Errors not caused by parameter block error.
11	Parameter block errors.

When a task resumes execution after making a CMR request, one of the above types of errors is reflected in bits 7-6 of register D0. The specific error can then be determined by examining the entire low order byte of register D0.

The I/O handlers return an error code within the event message returned in the ASQ or the return status buffer.

Type 00 errors are summarized in Table A-1.

Type 10 errors are subdivided into four categories. Table A-2 summarizes all of the type 10 errors, showing the four categories.

Type 11 errors indicate which field of the parameter block was in error. The field number would reside in bits 5-0 of the error code byte. The fields of a parameter block are numbered sequentially starting at 1, and do not correspond to the relative byte offset of a field. The fields of the parameter block for each CMR routine are listed and described in paragraph A.2.3.

TABLE A-1. Summary of Type 00 Errors

ERROR CODE	ERROR DESCRIPTION
2	Parameter block address not in requesting task's address space.
3	Target task does not exist.
6	Requested function has already been performed for the channel, and duplicate request not allowed.
8	Memory space not available.
9	Requested function can only be requested by a system task.
B	Request conflicts with existing CCB.

TABLE A-2. Summary of Type 10 Errors

ERROR CODE	ERROR DESCRIPTION
$81	Channel not attached to caller.
$82	Channel attached to another task.
$83	Channel offline.
$84	Channel busy.
$85	Invalid call for this channel type.
$C1	Bad function code.
$C6	Bad ASQ length (attach).
$C7	Service vector error (attach).
$C9	Bad channel base address (allocate).
$CB	Bad vector number (allocate).
$CC	Bad polling (hardware) priority (allocate).
$CE	Bad polling TBL count (allocate).
$CF	Bad polling offset (allocate). or Bad reset byte offset (allocate).

NOTE: $8x codes can be generated by channel management requests or by I/O requests to standard or nonstandard type channels.

A.2.13 I/O Completion Event for Standard Channels

Offset	Length	Field
0	1	Length
1	1	Code (1)
2	1	Event type
3	1	CCB key
4	1	ID (DCB address)
8	2	Status value (device dependent)
10		Unsolicited status/configuration information

Event Type

$70 = Normal
$71 = Halt
$80 = Unsolicited channel event
$FF = Unsolicited channel event

Status Value

$70 = Variable
$71 = (/0) - Command halted
0 - Nothing to halt
$80 = 0 - Device status (DCB address given)
1 - Device status (Channel ID given)
$FF = 0 - Channel reset
$FF - Channel down

THIS PAGE INTENTIONALLY LEFT BLANK.

APPENDIX B

DRIVER CALLS TO RMS68K (TRAP #0)

B.1 INTRODUCTION

This appendix describes several routines within RMS68K which may be called by any module executing in supervisor mode. Accordingly, they are intended primarily for use by user-written Executive procedures, such as input/output device drivers. They are not intended for use by modules executing in user mode.

B.2 SUMMARY DESCRIPTIONS OF THE ROUTINES

The routines described in this appendix provide operating services in four categories: task-related services, memory management services, semaphore services, and utility services. With a single exception, all of the services are available at only one level within an operating environment subdivided functionally into three levels.

At the highest level, tasks (which may be user or system tasks) are applications programs which execute only in the user mode of the MC68000. Tasks request services from the operating system through TRAP #n calls, where #n may be #1 through #15. Appendix C lists the services provided by each TRAP.

A TRAP is also called a programmed interrupt. TRAP #1 interrupts are preceded by loading register D0 with the service request code of the required service. This code is used to compute a jump through a vector table. In the cases of TRAPs #2 through #4, the programmed interrupt is handled by an interrupt processor routine called a "server". There is only one server associated with each TRAP. The user mode program provides a parameter block address. By accessing this block, the server determines the nature of the requested service. Even though the server is a component of the operating system, it is classified as a task. Therefore, it executes in user mode. The server coordinates multiple requests of the same class from various user and system tasks at a logical level of management. According to its priority scheme, it requests various operating system services on a one-at-a-time basis.

All tasks in VERSAdos, including the servers associated with TRAPs #2, #3, and #4, may execute TRAP #1 interrupts. There is no architectural requirement to proceed toward the kernel in strict TRAP number sequence. By issuing TRAP #1 interrupts, the servers most commonly access I/O driver modules, which are one example of a class of modules called Executive procedures.

Executive procedures execute only in supervisor mode as part of the RMS68K Executive or kernel. Accordingly, they may issue TRAP #0 interrupts. Executive procedures share hardware/software resources with user tasks on a managed basis. Examples of these resources include memory space, execution time, specialized conversion functions, and I/O to and from shareable and nonshareable devices. The Executive procedures written by users are most commonly I/O drivers.

If the driver is structured to permit a user task's own-code exit subroutine to be named -- for example, in association with specialized data conversion after a read operation -- the own-code subroutine is classed as a "dynamic extension" of the driver. Accordingly, it also executes in supervisor mode, even though it is provided as part of a user task. Because the user task's own-code subroutine executes in supervisor mode, it too may issue TRAP #0 interrupts if appropriate, even though it resides at the highest level of the system complex. The own-code exit subroutine is described in the RAD1 Device Driver Software User's Manual.

Many of the TRAP #0 routines described below provide adjunctive services associated with I/O resources and are "notification method" specific. When a user task requests an I/O operation, it elects to be notified of the completion of that operation by one of two methods -- queue-event or wakeup.

B.2.1 Mnemonic Abbreviations

In the descriptions of the routines in this appendix, the following mnemonics are used:

ASR	Asynchronous Service Request
ASQ	Asynchronous Service Queue
GST	Global Segment Table
MMIO	Memory Mapped Input/Output
MMU	Memory Management Unit
PAT	Periodic Activation Table
TCB	Task Control Block
TST	Task Segment Table
UST	User Semaphore Table

Refer to the M68000 Family Real-Time Multitasking Software User's Manual for descriptions of the tables and control blocks referenced above.

B.2.2 Task-Related Services

Task-related services encompass both periodic activation of an Executive procedure and post-event notification of user tasks.

EXRQPA

An Executive procedure may be scheduled or descheduled for activation on a periodic basis.

EXQEVENT

An I/O event may be queued to its associated task. The READY function, described below, is implicitly included.

PAUSE

While waiting for the completion of an I/O operation, an Executive procedure may relinquish its allocated time.

READY

An Executive procedure may change the state of a task from DORMANT to READY, in preparation for reactivation in accordance with the task's elected notification method. The READY function is implicit in both EXQEVENT and in WAKEUP.

WAKEUP

A task may be reactivated upon completion of a requested service. The READY function, described above, is implicitly included.

B.2.3 Memory Management Services

Memory management services encompass the allocation and deallocation of physical memory, the conversion of logical-to-physical memory address, and address look-up services for various tables maintained by the operating system.

FNDGSEG

The address of a shared local or global memory segment may be retrieved from the GST maintained by the operating system.

FNDTSEG

The address of a user or system task segment may be retrieved from the TST maintained for each running task.

GETTCB

The address of a Task Control Block may be retrieved from the TCB Table maintained by the operating system.

LOGPHY

The logical address of a memory segment may be converted to its physical address.

PAGEALOC

One or more pages of physical memory may be allocated to a task.

PAGEFREE

Page(s) of physical memory may be deallocated.

B.2.4 Semaphore Services

Semaphore services encompass both semaphore locater service and exclusive resource queueing service.

FNDUSEM

A task-associated semaphore may be located on a matching name basis or by association with the specified task.

PVSEM

A request for exclusive use of a resource may be queued or dequeued.

B.2.5 Utility Services

Utility services encompass the reporting of the current time of day and the saving of crash dump data.

RDTIMER

The time of day may be read and returned.

KILLER

The cause of system trouble may be saved and the system deliberately crashed.

B.3 SUMMARY DESCRIPTION OF CALL PROCEDURES

The structure of all TRAP #0 calls includes several classes of programmed operations. Prior to executing the call, register D0 is loaded with the service request code associated with each routine in the documentation appearing in Part 4 of this appendix. Depending on the routine being called, various other registers are loaded with data and addresses. The call itself is then executed in one of three ways. First, if the code section issuing the call is included in the same load module with the Executive, the fastest method of executing the call is to issue a Branch-to-Subroutine (BSR) instruction to the desired entry point name which has been declared external. The link loader will resolve the reference when the load module is created. Second, if the code section issuing the call is not to be included in the same load module with the Executive, the call can be executed by issuing the command TRAP #0. Third, also if the code section is not to be included in the same load module as the Executive, users can obtain the address of the routine using the ENTRY call (refer to paragraph G.6) during initialization, and then just call the routine as a subroutine. Finally, whichever call procedure is used, the commands following it are nearly always associated with a successful and an unsuccessful result, respectively. The command immediately following the call handles a successful result, except for KILLERTO and KILLER, which never return. It is shown in the examples as a short branch, because an unsuccessful result returns at the instruction following the call, plus two bytes. In one case, LOGPHY, a third return follows in the same manner as the second, with an offset of four bytes from the instruction following the call.

B.3.1 Separately Cataloged Routines

The call structure used by code sections to be cataloged separately from the Executive load module is typically:

```
MOVE.L <service-code>,D0
LEA    <param-table>,A0
TRAP   #0
BRA.S  <successful-result>    Return 1
<unsuccessful-result>         Return 2
```

The structure of the parameter table and register use differs with each TRAP #0 routine in question and is fully documented for each in Part 4.

Refer to paragraph G.6 for the ENTRY calling structure.

B.3.2 Integral Supervisor Routines

The call structure used by code sections to be included in the same load module with the Executive is typically:

```
LEA       <param-table>,A0
BSR       <entry-point>
BRA.S     <successful-result> Return 1
<unsuccessful-result>         Return 2
```

The structure of the parameter table and register use differs with each TRAP #0 routine in question and is fully documented for each in Section 4.

B.4 ALPHABETIC LISTING OF CALLING SEQUENCE

The TRAP #0 routines summarized in the preceding paragraphs are described in detail below. Several of the routines feature multiple entry points adapted to various needs. In the case of each entry point, alternate procedures are presented in accordance with the conventions described above.

All entry point names to be accessed by Branch-to-Subroutine instructions begin with the letters "SB...". Their associated preparation and return sequences appear after the illustration of the TRAP #0 version.

Following the detailed descriptions begins a condensed alphabetic listing of all TRAP #0 routines. The condition of all data and address registers is shown for each entry and alternate return. The appearance of a word preceded by a pound sign (e.g., # count, # entry) indicates that the register contains an integer binary value. The symbol "N/A" means "not applicable"; "N/C" means "no change". The angular brackets "<...>" enclose an item, known as a syntactic variable, that is replaced in a command line by one of a class of items it represents. Square brackets "[...]" enclose an item that is optional.

B.4.1 EXQEVENT

Upon its occurrence, an event may be queued from the Executive to a task. Three entry points are provided, selection of which is governed by the manner in which the target task is identified and by whether or not the Executive procedure is an interrupt handler.

Upon entry to EXQEVENT, registers D2 through D7 contain event descriptor data and event data itself. If additional space is needed for event data, registers A1 and A2 are considered continuations from D7. The byte structure of register D2 is $nnttcccc, where "nn" is the length of the event data in bytes, "tt" is the message type, and "cccc" are the first two bytes of the message. For a complete list of message types and their associated codes, refer to the M68000 Family Real-Time Multitasking Software User's Manual.

EXQEVENT features two sets of entry points, depending on how the task is identified and whether or not the caller is an interrupt handler. In the case of a caller that is not an interrupt handler, if the target task is identified by taskname and session number, the EXQEVENT entry points are EXQEVNTN and SBQEVNTN:

EXQEVNTN calling sequence:

```
                MOVE.L #11,D0       Service request code for EXQEVNTN
                LEA    TABLE,A0
                TRAP   #0
                BRA.S  <queued>     Return 1
                <not-queued>        Return 2
                .
                .
        TABLE   DC.L   <taskname>
                DC.L   <session-number>
```

SBQEVNTN calling sequence:

```
                LEA       TABLE,A0
                BSR       SBQEVNTN
                BRA.S     <queued>  Return 1
                <not-queued>        Return 2
```

If the target task is identified by a TCB, the EXQEVENT entry points are EXQEVNTT and SBQEVNTT:

EXQEVNTT calling sequence:

```
                MOVE.L    #23,D0    Service request code for EXQEVNTT
                LEA       <target-TCB>,A0
                TRAP      #0
                BRA.S     <queued>  Return 1
                <not-queued>        Return 2
```

SBQEVNTT calling sequence:

```
LEA          <target-TCB>,A0
BSR          SBQEVNTT
BRA.S        <queued>  Return 1
<not-queued>           Return 2
```

If the event is being queued by an interrupt handler executing at an interrupt level greater than 0, the target task is identified by a TCB and the entry points are EXQEVNTI and SBQEVNTI:

EXQEVNTI calling sequence:

```
MOVE.L       #24,D0    Service request code for EXQEVNTI
LEA          <target-TCB>,A0
TRAP         #0
BRA.S        <queued>  Return 1
<not-queued>           Return 2
```

SBQEVNTI calling sequence:

```
LEA          <target-TCB>,A0
BSR          SBQEVENTI
BRA.S        <queued>  Return 1
<not-queued>           Return 2
```

EXQEVENT returns to the instruction following the call if the event was queued successfully. If the event was not queued successfully, EXQEVENT returns to the instruction following the call, plus two bytes.

Upon exit from EXQEVENT, the registers remain as configured on entry. In addition, register A4 contains the ASQ address, and register A5 contains the target task's TCB address.

B.4.2 EXRQPA

An Executive procedure may be activated one time or periodically, and a periodically activated procedure may be deactivated. Both the characteristics of the activated procedure and the structure of the periodic activation request must be considered.

B.4.2.1 The Executive Procedure. Periodic activation is controlled by the system timer in the RMS68K Executive. An interrupt occurs at approximately 10-millisecond intervals, causing the Executive to scan the PAT or Executive procedures whose identities have been declared through calls to EXRQPA. Those procedures whose dormant intervals have expired are activated.

Periodically activated Executive procedures are classed as interrupt handlers. Accordingly, they must be associated with an interrupt level when their identities are declared through calls to EXRQPA. The interrupt level range must be between 1 and 6, inclusive.

The requested activation may specify one-time only or continuous activation modes. A single activation might be requested, for example, to test for time-out on an I/O device. Periodic activation on a continuous basis might be requested, for example, to poll an I/O device.

On entry to the activated Executive procedure, register D0 contains the number of intervals represented by this activation. The length of an interval is the number of milliseconds specified through the call to EXRQPA which declared this Executive procedure for activation. If the value of register D0 is greater than 1, it means that n-1 activations have been missed due to higher priority requests. In addition to register D0, register D1 contains the 32-bit activation identification value specified through the call to EXRQPA which declared this Executive procedure for activation. The contents of registers D0 and D1 are discussed in detail below. If the Executive procedure uses registers D2 through D7 or registers A2 through A7, they must be saved on entry. If a timer interrupt occurs within the Executive procedure, registers D0, D1, A0, and A1 will be saved and restored by the timer interrupt handler.

Exit from an activated Executive procedure is through an M68000 RT instruction after restoring any saved registers D2 through D7 and A2 through A7.

B.4.2.2 Activation Request Procedures. On entry to EXRQPA, register D0 contains the service request code. Register D1 expresses the periodic activation type code and interrupt level, as follows:

Bit No.	Value	Meaning
15	0	Request for new periodic activation.
	1	Cancel periodic activation request.
14	0	Request single activation.
	1	Request continuous activation.
13	0	If another activation request is currently outstanding which has the same activation ID as this request, cancel it before registering the new request. Only one such request will be cancelled.
	1	Do not cancel a request with the same ID before registering the new request. It will be possible to have more than one request with the same ID in effect at the same time. This option executes faster than the option with bit 13=0.
12-8	000000	Unused.
7-3	00000	Unused.
2-0	nnn	Interrupt level 1-6.

Register D2 contains a 32-bit value to serve as the request identification. The contents are not important except that they are unique. Register A0 contains the address of the procedure to be activated. Register A1 contains the activation interval expressed in milliseconds.

For additional information on interrupt levels, refer to the 16/32-Bit Microprocessor Programmer's Reference Manual.

There is only one set of entry points to EXRQPA, including EXRQPA and SBRQPA:

EXRQPA calling sequence:

```
        MOVE,L      #34,D0                 Service request code for EXRQPA
        MOVE.W      <options>,D1
        MOVE.L      <request-id>,D2
        LEA         <procedure>,A0
        MOVE.L      <interval>,A1
        TRAP        #0
        BRA.S       <accepted>             Return 1
        <not-accepted>                     Return 2
```

SBRQPA calling sequence:

```
        MOVE.W      <options>,D1
        MOVE.L      <request-id>,D2
        LEA         <procedure>,A0
        MOVE.L      <interval>,A1
        BSR         SBRQPA
        BRA.S       <accepted>             Return 1
        <not-accepted>                     Return 2
```

On exit, registers D0 and D1 are destroyed. Registers D2 through D7 and A0 through A7 remain as configured on entry.

B.4.3 FNDGSEG

FNDGSEG locates shared local and global segments, both of which are listed in the GST maintained by the Executive. The caller may locate an existing segment entry, obtain a pointer to an unused segment entry, or determine that the GST is full.

On entry to FNDGSEG, register D0 contains the service request code. Register D1 contains the segment attributes, as follows:

Bit No.	Value	Meaning
31-16	$0000	Unused.
15-14	00	Unused.
13	0	Segment is not local.
	1	Segment is shared locally.
12	0	Segment is not global.
	1	Segment is shared globally.
11-8	$0	Unused.
7-0	$00	Unused.

The only valid configurations of bits 12 and 13 are 01 and 10. The invalid configurations 00 and 11 will cause FNDSEG to return as though the named segment was not found, as described in exit conditions in the next paragraph. Register D2 contains the task's session number. Register A0 contains the segment name. There is only one set of entry points to FNDGSEG, including FNDGSEG and SBFNDGSG.

FNDGSEG calling sequence:

```
    MOVE.L    #9,D0                     Service request code for FNDGSEG
    MOVE.L    <segment-attributes>,D1
    MOVE.L    <session-number>,D2
    MOVE.L    <segment-name>,A0
    TRAP #0
    BRA.S     <found>                   Return 1
    <not found>                         Return 2
```

SBFNDGSG calling sequence:

```
    MOVE.L    <segment-attributes>,D1
    MOVE.L    <session-number>,D2
    MOVE.L    <segment-name>,A0
    BSR       SBFNDGSG
    BRA.S     <found>                   Return 1
    <not-found>                         Return 2
```

On exit, FNDGSEG returns to the instruction following the call if the named segment was found. If it was not found, or if its segment attributes specified an illegal combination in bits 12 and 13, the return will occur at the next instruction following the call, plus two bytes. The configuration of register A0 is different for each of these alternate returns. In both returns, however, register D0 contains the updated count of the total number of entries in the GST. Registers D1 and D2 remain as configured on entry. The contents of registers D3, A1, A2, and A3 are destroyed.

A return to the instruction following the call indicates that the named segment was found in the GST. Register A0 points to the entry by entry number.

A return to the instruction following the call, plus two bytes, indicates that (1) the named segment was not found (also caused by illegal segment attributes specification), and (2) either a new entry is provided or the GST is full. A nonzero value returned in register A0 points to the next available table entry by entry number. A zero value indicates that the GST is full.

B.4.4 FNDTSEG

FNDTSEG locates a user or system task segment in the TST associated with the task, and returns both the TST entry pointer and an offset to the segment description entry within the TST.

On entry, register D0 contains the service request code. Register D7 contains the segment name.

There are only two entry points to FNDTSEG, including FNDTSEG and SBFNDSEG.

FNDTSEG calling sequence:

```
    MOVE.L    #7,D0                Service request code for FNDTSEG
    MOVE.L    <session-number>,D2
    MOVE.L    <segment-name>,D7
    LEA       <TST-address>,A0
    TRAP #0
    BRA.S     <found>              Return 1
    <not-found>                    Return 2
```

On exit, FNDTSEG will return to the instruction following the call if the named segment was found. If the named segment was not found, FNDTSEG will return to the instruction following the call, plus two bytes. In both cases, register D0 is destroyed. Register D5 varies with each return. Registers D7 and A0 remain as configured on entry.

If the named segment was found, register D5 contains the offset within the TST where the named segment was found.

If space remains available in the TST while the named segment was not found, register D5 contains the offset within the TST of the first empty table entry. Register D5 will contain the value zero if the TST is full.

B.4.5 FNDUSEM

A task semaphore entry may be located in the UST maintained by the Executive by either of two locator methods. The semaphore may be located by its name if it is known. If not, any semaphore owned by the designated task may be located by a match on the target task's TCB.

On entry, register D0 contains the service request code. Register A0 controls the method of search selected. If register A0 contains the value zero, the search will locate any semaphore associated with the specified TCB. If A0 is not equal to zero, it will be matched against the user task semaphore names in the UST. Register A4 contains the target task's TCB address.

There is only one set of entry points to FNDUSEM, including FNDUSEM and SBFNDSEM.

FNDUSEM calling sequence:

```
    MOVE.L    #12,D0               Service request code for FNDUSEM
    MOVE.L    <selection>,A0
    LEA       <TCB-address>,A4
    TRAP #0
    BRA.S     <found>              Return 1
    <not-found>                    Return 2
```

SBFNDSEM calling sequence:

```
MOVE.L    <selection>,A0
LEA       <TCB-address>,A4
BRS       SBFNDSEM
BRA.S     <found>              Return 1
<not-found>                    Return 2
```

On exit, FNDUSEM will return to the first location following the call if a match has been found in the UST for either the specified semaphore name or the task's TCB. FNDUSEM will return to the first location following the call, plus two bytes if a match has not been found.

If a match has not been found, FNDUSEM will indicate either the next available entry in the UST or that the UST is full. Registers D0 and D2 vary, depending on the return selected. Registers D1, D3, and D4 are destroyed. Register A0 varies, depending on the return selected. Register A4 remains as configured on entry.

If a match has been found on either the user semaphore or the task's TCB specified with the call, register D0 contains the table entry number where a semaphore or TCB match was found. Register D2 contains the current number of entries in the UST. Register A0 contains the semaphore name.

If a match was not found on either the specified semaphore name or the specified TCB, according to the request form selected on entry, the configuration of the registers depends on whether or not an unused entry was available in the UST. In both cases, register D2 contains the updated or current number of UST entries, and register A0 contains the value zero.

If an unused entry were available, register D0 contains the entry number of the available entry. Register D1 contains the entry number of the primary reference to a semaphore of the same name, if the caller's reference is a secondary or successive such reference. Multiple references occur when a semaphore is shared by more than one task. A multiply referenced semaphore appears only once, when initially declared. Thereafter, other references create only pointers to the original reference. If the current reference is unique, register D1 will contain the value zero.

If no unused entry is available in the UST, both registers D0 and D1 contain the value zero.

B.4.6 GETTCB

GETTCB will locate a TCB address in the linked list of TCBs maintained by the Executive, based on the taskname and session number of the task.

On entry, register D0 contains the service request code. Register A0 contains the address of the target taskname and session number.

GETTCB provides two sets of entry points, depending on caller type. GETTCB is the principal security device within the operating system. Because user tasks are permitted to access resources used by other tasks only if the session numbers are the same, GETTCB verifies its target task identification procedure for user tasks as compared to system tasks. If the caller is a user task, the session number supplied with the target taskname is ignored. The calling task's session number is supplied automatically. Thus, a call to locate another user task with a different session number will always fail after a search of the linked list of TCBs. If the caller is a system task, however, the session number supplied with the target taskname will be honored. Thus, the call to locate other system tasks, as well as any user task, will succeed. One set, consisting of GETTCB and SBGETTCB, is used to retrieve the target TCB address if the caller is a user task:

GETTCB calling sequence:

```
         MOVE.L    #6,D0                 Service request code for GETTCB
         LEA       TABLE,A0
         TRAP      #0
         BRA.S     <found>               Return 1
         <not-found>                     Return 2
         .
         .
TABLE    DC.L      <taskname>
         DC.L      <session-number>
```

SBGETTCB calling sequence:

```
         LEA       TABLE,A0
         BSR       SBGETTCB
         BRA.S     <found>               Return 1
         <not-found>                     Return 2
         .
         .
TABLE    DC.L      <taskname>
         DC.L      <session-number>
```

The second set of entry points, GTXTCB and SBGTXTCB, is used to retrieve the target TCB address if the caller is a system task.

GTXTCB calling sequence:

```
MOVE.L    #13,D0                Service request code for  GTXTCB
MOVE.L    <taskname>,A0
MOVE.L    <session-number>,D1
TRAP #0
BRA.S     <found>               Return 1
<not-found>                     Return 2
```

SBGTXTCB calling sequence:

```
MOVE.L    <taskname>,A0
MOVE.L    <session-number>,D1
BSR       SBGTXTCB
BRA.S     <found>             Return 1
<not-found>                   Return 2
```

On exit, GETTCB will return to the instruction following the call if the indicated TCB address was found. If it was not found, GETTCB will return to the instruction following the call, plus two bytes. In both cases, registers D0, D1, and D2 are destroyed.

If the specified user task or system task TCB was found, the TCB address is returned in register A0. Register A6 remains configured as on entry.

B.4.7 KILLER

KILLER saves the cause of system trouble and crashes the operating system. Refer to the M68000 Family Real-Time Multitasking Software User's Manual for information on how to analyze a system crash.

KILLER provides a single set of entry points, including KILLERTO and KILLER.

KILLERTO calling sequence:

```
MOVE.L    #32,D0                Service request code for KILLERTO
TRAP      #0
```

KILLER calling sequence:

```
BSR KILLER
```

There is no return from either KILLERTO or KILLER because the system is no longer operational.

B.4.8 LOGPHY

A logical base address, beginning from either an even or an odd number, may be converted to its physical address. In addition, the manner of return from the service routine indicates the location of the specified memory "window" with respect to the boundaries of the specified memory segment.

A memory "window" is a stream of bytes beginning at a specified relative or logical base address and continuing contiguously and incrementally through a specified number of bytes. The window may reside totally within a specified segment of memory, it may straddle a segment boundary, or it may be completely outside of the specified segment.

Upon entry to LOGPHY, register D0 contains the service request code. Register D5 contains the length of the window in bytes. Register D6 contains the logical beginning address of the window. Register A0 contains the calling task's TST address.

LOGPHY provides two sets of entry points, depending on whether the supplied logical address is an even or an odd number. The entry points LOGPHY and SBLOGPHY clear bit zero of the returned physical address to zero, so that conversion always yields an even result.

LOGPHY calling sequence:

```
MOVE.L    #8,D0                    Service request code for LOGPHY
MOVE.L    <logical-address>,D6
MOVE.L    <TST-address>,A0
TRAP #0
BRA.S     <good-return>                      Return 1
BRA.S     <part-contained>                   Return 2
[<not-contained>] [MMIO-segment>]            Return 3
```

SBLOGPHY calling sequence:

```
MOVE.L    <logical-address>,D6
MOVE.L    <TST-address>,A0
BSR       SBLOGPHY
BRA.S     <good-return>                      Return 1
BRA.S     <part-contained>                   Return 2
[<not-contained>] [MMIO-segment>]            Return 3
```

The entry points LOGPHYO and SBLOGPHO will return both even and odd results after conversion.

LOGPHYO calling sequence:

```
MOVE.L    #26,D0                   Service request code for LOGPHYO
MOVE.L    <logical-address>,D6
MOVE.L    <TST-address>,A0
TRAP #0
BRA.S     <good-return>                      Return 1
BRA.S     <part-contained>                   Return 2
[<not-contained>] [<MMIO-segment>]           Return 3
```

SBLOGPHO calling sequence:

```
MOVE.L    <logical-address>,D6
MOVE.L    <TST-address>,A0
BSR       SBLOGPHO
BRA.S     <contained>                     Return 1
BRA.S     <part-contained>                Return 2
[<not-contained>] [<MMIO-segment>]        Return 3
```

On exit, LOGPHY features three returns, depending on the location of the specified window within, partially within, or completely outside of the specified memory segment. In the case of all three returns, registers D0, D3, and D4 are destroyed. Registers D5 and D6 vary with each return. Register A0 remains configured as on entry.

A return to the location following the call indicates that the specified window lies completely within the specified memory segment. Register D5 contains the offset within the TST to the MMU entry. Register D6 contains the physical beginning address of the window.

A return to the location following the call, plus two bytes, indicates that the specified window lies partially within the specified segment. Register D5 contains the offset within the TST to the MMU entry for the portion of the window contained within the specified segment. Register D6 remains as configured on entry.

A return to the location following the call, plus four bytes, indicates that the specified window lies completely outside of the specified segment. Register D5 contains the offset within the TST of the first empty entry in the MMU, or it points to the MMIO segment, or it is a zero value indicating no MMU or MMIO segment. Register D6 remains as configured on entry. In all three cases, the entry may be retrieved by the addressing structure 0(A0,D5).

B.4.9 PAGEALOG

PAGEALOC allocates physical memory in 256-byte pages. A variety of memory types and options may be specified. One or more pages may be requested by specified count, or simply the largest block of contiguous pages which is currently available. The allocation may be requested from a specified base address, and the specified block may be MMIO, ROM, or RAM.

On entry, register D0 contains the service request code. Register A0 contains a 2-word description of the memory type and options desired, and the number of pages requested:

Bit No.	Value	Meaning
31-27	00000	Unused.
26	0	Do not wait if memory not available.
	1	Wait if memory not available.
25	0	Allocate specified number of pages.
	1	Allocate largest block available.
24	0	Allocate where found.
	1	Allocate at specified address.
23	0	Block is RAM.
	1	Block is ROM or MMIO.
22-20	nnn	Memory types (refer to NOTE below).
19-16	nnnn	Partition numbers (refer to NOTE below).
15-0	$nnnn	Number of pages to allocate.

NOTE

Memory type and partition number are discussed in detail in M68000 Family Real-Time Multitasking Software User's Manual, Chapter 3, "GTSEG" call.

If bit 24 of register A0 is 1, then register A1 is the 24-bit address from which the allocation of the block of page(s) should take place. Otherwise, register A1 equals zero. Register A6 contains the address of the caller's TCB.

PAGEALOC features only one set of entry points, including PAGEALOC and SBPAGAL:

PAGEALOC calling sequence:

```
MOVE.L    #4,D0                 Service request code for PAGEALOC
MOVE.L    <description>,A0
MOVE.L    <physical-address>,A1
TRAP #0
BRA.S     <allocated>           Return 1
<not-allocated>                 Return 2
```

SBPAGAL calling sequence:

```
MOVE.L    <description>,A0
MOVE.L    <physical-address>,A1
BSR       SBPAGAL
BRA.S     <allocated>           Return 1
<not-allocated>                 Return 2
```

On exit, PAGEALOC returns to the instruction following the call if the page allocation request succeeded. If page allocation was not successful, return is to the instruction following the call, plus two bytes.
If page allocation was successful as requested, register D1 contains a 1-byte value indicating the partition number within which the allocation occurred, as follows:

Bit No.	Value	Meaning
31-16	$0000	Unused.
15-8	$00	Unused.
7-4	nnnn	Memory type.
3-0	nnnn	Memory partition number.

Register D2 echoes the number of pages allocated if a specified number was requested, or states the number allocated if the largest-block option was specified with the call. Register A0 contains the physical base address of the allocated block. Registers A1 and A2 were destroyed, and register A6 remains configured as on entry.

If the allocation failed, registers D0, D1, D2, A0, A1, and A2 are assumed to be destroyed.

B.4.10 PAGEFREE

PAGEFREE deallocates one or more 256-byte pages of memory. If more than one page of memory is freed, the second and successive pages are incrementally contiguous to the first page freed.

On entry, register D0 contains the service request code. Register D1 contains the number of pages to be freed. Register A0 contains the starting address of the first page to be freed.

PAGEFREE features a single set of entry points, including PAGEFREE and SBPGFR.

PAGEFREE calling sequence:

```
    MOVE.L    #5,D0                 Service request code for PAGEFREE
    MOVE.L    <page-count>,D1
    MOVE.L    <start-address>,A0
    TRAP      #0
    BRA.S     <pages-freed>         Return 1
    <not-freed>                     Return 2
```

SBPGFR calling sequence:

```
    MOVE.L    <page-count>,D1
    MOVE.L    <start-address>,A0
    BSR       SBPGFR
    BRA.S     <pages-freed>         Return 1
    <not-freed>                     Return 2
```

On exit, PAGEFREE returns to the instruction following the call if the deallocation was successful. It returns to the instruction following the call, plus two bytes, if the deallocation was not successful. In both cases, registers D0, D1, D2, A0, A1, and A2 are destroyed.

B.4.11 PAUSE

The currently running Executive procedure issues a call to PAUSE when it must wait on the occurrence of an external event before continuing. As a consequence of issuing the call, the task which called the currently running Executive procedure will be placed in a "return-to-Executive" state. Control will then be given to the dispatcher.

Control will return to the instruction following the call to PAUSE after the suspended user task is moved to the READY state by some other Executive procedure. (Refer to the READY routine, B.4.14.)

On entry, register D0 contains the service request code. Register A6 contains the target task's TCB address. Register A6 is loaded with the running task's TCB address whenever a TRAP #1 directive is called. Register A6 should not be modified by any Executive procedure executing as part of a TRAP #1 directive.

PAUSE features a single set of entry points, including PAUSE and SBPAUSE.

PAUSE calling sequence:

```
    MOVE.L    #14,D0                 Service request code for PAUSE
    MOVE.L    <TCB-address>,A6
    TRAP #0
```

SBPAUSE calling sequence:

```
    MOVE.L    <TCB-address>,A6
    BSR       SBPAUSE
```

On exit, PAUSE returns to the instruction following the call. Register D0 is destroyed and register A6 is configured as on entry.

B.4.12 PVSEM

PVSEM manages the queuing and dequeuing of tasks requesting exclusive use of a system resource. PVSEM should not be called from an interrupt service routine.

On entry, register D0 contains the service request code. Register A0 contains the address of the semaphore. The semaphore is a 6-byte field constructed as illustrated at "SEMAPH" in the call sequence below. Register A6 contains the running task's TCB address. The configuration of register A6 occurs whenever a TRAP #1 directive is called. A6 should not be modified by any Executive procedure executing as part of a TRAP #1 directive.

PVSEM features two sets of entry points, one each for requesting and for releasing exclusive use of a system resource.

Request for exclusive use of a resource is made through entry points PSEM4 and SBP.

PSEM4 calling sequence:

```
         MOVE.L    #1,D0                Service request code for PSEM4
         LEA       SEMAPH,A0
         MOVE.L    <TCB-address>,A6
         TRAP      #0
         .
         .
SEMAPH   DC.W      #0
         DC.L      #0
```

SBP calling sequence:

```
         LEA       SEMAPH,A0
         MOVE.L    <TCB-address>,A6
         BSR       SBP
         .
         .
SEMAPH   DC.W      #0
         DC.L      #0
```

On entry through either of the above calls, PVSEM verifies the availability status of the requested resource.

If the resource is available, its availability status is reset to "BUSY", and control is returned to the caller. The caller is then free to access the requested resource. On completion of its use, the caller releases the resource through the calls illustrated below.

If the requested resource is already in "BUSY" status, PVSEM queues the caller into a linked list of waiting tasks. Return is not made to the caller. Instead, control is transferred to the dispatcher. When the task queued ahead of the current caller has released exclusive use of the resource, control returns to the next caller in the linked list.

Release of an exclusive resource is made through entry points VSEM4 and SBV.

VSEM4 calling sequence:

```
         MOVE.L    #2,D0                Service request code for VSEM4
         LEA       SEMAPH,A0
         MOVE.L    <TCB-address>,A6
         TRAP      #0
         .
         .
SEMAPH   DC.W      #0
         DC.L      #0
```

SBV calling sequence:

```
        LEA       SEMAPH,A0
        MOVE.L    <TCB-address>,A6
        BSR       SBV
        .
        .
SEMAPH  DC.W      #0
        DC.L      #0
```

On exit, return is made to the next instruction following the call. The 2-byte initial field in "SEMAPH", above, contains a reserved high-order bit which is always zero on return to the caller. The remaining 15 bits are an integer count of the number of requests for this same resource which are queued in the WAIT list behind the current caller. The 15-bit value expresses the WAIT count as a simple negative; that is, a 15-bit zero minus one equals $7FFF. The 4-byte field in "SEMAPH", above, is the address of the first TCB in the WAIT list.

B.4.13 RDTIMER

The caller may obtain the time of day from the system timer in the Executive.

On entry, register D0 contains the service request code. RDTIMER features a single set of entry points, RDTIMER and SBRDTIM.

RDTIMER calling sequence:

```
        MOVE.L    #28,D0     Service request code for RDTIMER
        TRAP      #0
```

SBRDTIM calling sequence:

```
        BSR       RDTIMER
```

On exit, control returns to the instruction following the call. Register D0 contains the timer contents; register D1 contains the time of day expressed in milliseconds.

The contents of D0 may be used to calculate microseconds. The timer is loaded initially with two 8-bit values. Bits 15-8 contain the value of TIMEINTV*4-1; bits 7-0 contain the value of CLOCKFRQ/4-1. TIMEINTV and CLOCKFRQ are system generation parameters. Subtracting the current value from the initial setting and separating the two values will permit the caller to calculate the time when the timer was read. The precision of the result is controlled by the setting of CLOCKFRQ at system generation.

B.4.14 READY

The calling Executive procedure causes the designated target task to be moved to the READY state.

On entry, register D0 contains the service request code. Register A0 contains the address of the target task's TCB.

READY features a single set of entry points, READY and SBREADY.

READY calling sequence:

```
         MOVE.L    #3,D0                   Service request code for READY
         LEA       <TCB-address>,A0
         TRAP      #0
```

SBREADY calling sequence:

```
         LEA       <TCB-address>,A0
         BSR       SBREADY
```

On exit, READY returns to the instruction following the call. Register D0 is destroyed. Register A0 is configured as on entry. No arguments are returned.

B.4.15 WAKEUP

On the occurrence of an event that is associated with a task which has elected wakeup notification method, the event may be moved to the READY state.

If the task is not presently in the WAIT state, WAKEUP will set the status of the event to wakeup-pending. At a future time when the task places itself in WAIT state, it will be waked up immediately.

On entry, register D0 contains the service request code. Register A0 contains the target task's TCB address. WAKEUP features a single set of entry points, including WAKEUP and SBWAKEUP.

WAKEUP calling sequence:

```
         MOVE.L    #22,D0                  Service request code for WAKEUP
         MOVE.L    <TCB-address>,A0
         TRAP      #0
```

SBWAKEUP calling sequence:

```
         MOVE.L    <TCB-address>,A0
         BSR       SBWAKEUP
```

On exit, control resumes at the instruction following the call. Register D0 is destroyed. Register A0 remains configured as on entry.

TRAP #0 CALLING SEQUENCE SUMMARY

NAME: EXQEVNTI SBQEVNTI
FUNCTION: Queue event to target task; caller is an interrupt routine (level greater than 0).
PARA. B.4.1

	ENTRY	EXIT		
	CALL	RETURN 1	RETURN 2	RETURN 3
D0	#24	destroyed	destroyed	
D1		destroyed	destroyed	
D2	event-data	destroyed	destroyed	
D3	[event-data]	N/C	N/C	
D4	[event-data]	N/C	N/C	
D5	[event-data]	destroyed	destroyed	
D6	[event-data]	destroyed	destroyed	
D7	[event-data]	destroyed	destroyed	
A0	<TCB>	destroyed	destroyed	
A1	[event-data]	destroyed	destroyed	
A2	[event-data]	N/C	N/C	
A3		destroyed	destroyed	
A4		<ASQ>	<ASQ>	
A5		<TCB>	<TCB>	
A6		N/C	N/C	
A7		N/C	N/C	

NAME: EXQEVNTN SBQEVNTN
FUNCTION: Queue event to target task; caller is not an interrupt routine.
PARA. B.4.1

	ENTRY	EXIT		
	CALL	RETURN 1	RETURN 2	RETURN 3
D0	#11	destroyed	destroyed	
D1	event-data	destroyed	destroyed	
D2	[event-data]	N/C	N/C	
D3	[event-data]	N/C	N/C	
D4	[event-data]	destroyed	destroyed	
D5	[event-data]	destroyed	destroyed	
D6	[event-data]	destroyed	destroyed	
D7	[event-data]	destroyed	destroyed	
A0	<task-ID>	destroyed	destroyed	
A1	[event-data]	destroyed	destroyed	
A2	[event-data]	N/C	N/C	
A3		destroyed	destroyed	
A4		<ASQ>	<ASQ>	
A5		<TCB>	<TCB>	
A6		N/C	N/C	
A7		N/C	N/C	

NAME:	FUNCTION:	PARA.
EXQEVNTT SBQEVNTT	Queue event to target task; caller is not an interrupt routine.	B.4.1

	ENTRY	EXIT		
	CALL	RETURN 1	RETURN 2	RETURN 3
D0	#23	destroyed	destroyed	
D1		destroyed	destroyed	
D2	event-data	destroyed	destroyed	
D3	[event-data]	N/C	N/C	
D4	[event-data]	N/C	N/C	
D5	[event-data]	destroyed	destroyed	
D6	[event-data]	destroyed	destroyed	
D7	[event-data]	destroyed	destroyed	
A0	<TCB>	destroyed	destroyed	
A1	[event-data]	destroyed	destroyed	
A2	[event-data]	N/C	N/C	
A3		destroyed	destroyed	
A4		<ASQ>	<ASQ>	
A5		<TCB>	<TCB>	
A6		N/C	N/C	
A7		N/C	N/C	

NAME:	FUNCTION:	PARA.
EXRQPA SBRQPA	Request periodic activation/ deactivation of an Executive procedure.	B.4.2

	ENTRY	EXIT		
	CALL	RETURN 1	RETURN 2	RETURN 3
D0	#34	destroyed	destroyed	
D1	<options>	destroyed	destroyed	
D2	<request-ID>	N/C	N/C	
D3		N/C	N/C	
D4		N/C	N/C	
D5		N/C	N/C	
D6		N/C	N/C	
D7		N/C	N/C	
A0	<procedure>	N/C	N/C	
A1	<interval>	destroyed	destroyed	
A2		N/C	N/C	
A3		N/C	N/C	
A4		N/C	N/C	
A5		N/C	N/C	
A6		N/C	N/C	
A7		N/C	N/C	

NAME: FNDGSEG SBFNDGSG

FUNCTION: Locate shared local and global segment entries in Global Segment Table.

PARA. B.4.3

	ENTRY	EXIT		
	CALL	RETURN 1	RETURN 2	RETURN 3
D0	#9	# count	# count	
D1	<seg-attrib>	N/C	N/C	
D2	<session>	N/C	N/C	
D3		destroyed	destroyed	
D4		N/C	N/C	
D5		N/C	N/C	
D6		N/C	N/C	
D7		N/C	N/C	
A0	<seg-name>	# entry	# next/#0	
A1		destroyed	destroyed	
A2		destroyed	destroyed	
A3		destroyed	destroyed	
A4		N/C	N/C	
A5		N/C	N/C	
A6		N/C	N/C	
A7		N/C	N/C	

NAME: FNDTSEG SBFNDSEG

FUNCTION: Locate task segment entry in Task Segment Table.

PARA. B.4.4

	ENTRY	EXIT		
	CALL	RETURN 1	RETURN 2	RETURN 3
D0	#7	destroyed	destroyed	
D1		N/C	N/C	
D2	<session>	N/C	N/C	
D3		N/C	N/C	
D4		N/C	N/C	
D5		# offset	# next/#0	
D6		N/C	N/C	
D7		N/C	N/C	
A0	<seg-name>	N/C	N/C	
A1	<TST-addr>	N/C	N/C	
A2		N/C	N/C	
A3		N/C	N/C	
A4		N/C	N/C	
A5		N/C	N/C	
A6		N/C	N/C	
A7		N/C	N/C	

NAME: FNDUSEM SBFNDSEM	FUNCTION: Locate task semaphore entry in User Semaphore Table.			PARA. B.4.5
	ENTRY	EXIT		
	CALL	RETURN 1	RETURN 2	RETURN 3
D0	#12	# entry	# entry/#0	
D1		destroyed	destroyed	
D2		# count	# count/#0	
D3		destroyed	destroyed	
D4		destroyed	destroyed	
D5		N/C	N/C	
D6		N/C	N/C	
D7		N/C	N/C	
A0	<selection>	<sem-name>	#0	
A1		N/C	N/C	
A2		N/C	N/C	
A3		N/C	N/C	
A4	<TCB-addr>	N/C	N/C	
A5		N/C	N/C	
A6		N/C	N/C	
A7		N/C	N/C	

NAME: GETTCB SBGETTCB	FUNCTION: Locate TCB address in TCB table (user tasks only).			PARA. B.4.6
	ENTRY	EXIT		
	CALL	RETURN 1	RETURN 2	RETURN 3
D0	#6	destroyed	destroyed	
D1		destroyed	destroyed	
D2		N/C	N/C	
D3		N/C	N/C	
D4		N/C	N/C	
D5		N/C	N/C	
D6		N/C	N/C	
D7		N/C	N/C	
A0	<table>	<TCB-addr>	destroyed	
A1		destroyed	destroyed	
A2		N/C	N/C	
A3		N/C	N/C	
A4		N/C	N/C	
A5		N/C	N/C	
A6		N/C	N/C	
A7		N/C	N/C	

NAME: GTXTCB SBGTXTCB	FUNCTION: Locate TCB address in TCB Table (system tasks only).			PARA. B.4.6
	ENTRY	EXIT		
	CALL	RETURN 1	RETURN 2	RETURN 3
D0	#13	destroyed	destroyed	
D1	<session>	destroyed	destroyed	
D2		N/C	N/C	
D3		N/C	N/C	
D4		N/C	N/C	
D5		N/C	N/C	
D6		N/C	N/C	
D7		N/C	N/C	
A0	<task-name>	<TCB-addr>	destroyed	
A1		destroyed	destroyed	
A2		N/C	N/C	
A3		N/C	N/C	
A4		N/C	N/C	
A5		N/C	N/C	
A6		N/C	N/C	
A7		N/C	N/C	

NAME: KILLERTO KILLER	FUNCTION: Save cause of system trouble and crash system.			PARA. B.4.7
	ENTRY	EXIT		
	CALL	RETURN 1	RETURN 2	RETURN 3
D0	#32	N/A		
D1		N/A		
D3		N/A		
D4		N/A		
D5		N/A		
D6		N/A		
D7		N/A		
A0		N/A		
A1		N/A		
A2		N/A		
A3		N/A		
A4		N/A		
A5		N/A		
A6		N/A		
A7		N/A		

NAME: LOGPHY SBLOGPHY	FUNCTION: Convert logical to physical address (even numbers only).			PARA. B.4.8
	ENTRY	EXIT		
	CALL	RETURN 1	RETURN 2	RETURN 3
D0	#8	destroyed	destroyed	destroyed
D1		N/C	N/C	N/C
D2		N/C	N/C	N/C
D3		destroyed	destroyed	destroyed
D4		destroyed	destroyed	destroyed
D5		# offset	# offset	# offset/#0
D6	<logical-addr>	<phys-addr>	N/C	N/C
D7		N/C	N/C	N/C
A0	<TST-addr>	N/C	N/C	N/C
A1		N/C	N/C	N/C
A2		N/C	N/C	N/C
A3		N/C	N/C	N/C
A4		N/C	N/C	N/C
A5		N/C	N/C	N/C
A6		N/C	N/C	N/C
A7		N/C	N/C	N/C

NAME: LOGPHYO SBLOGPHO	FUNCTION: Convert logical to physical address (odd numbers only).			PARA. B.4.8
	ENTRY	EXIT		
	CALL	RETURN 1	RETURN 2	RETURN 3
D0	#26	destroyed	destroyed	destroyed
D1		N/C	N/C	N/C
D2		N/C	N/C	N/C
D3		destroyed	destroyed	destroyed
D4		destroyed	destroyed	destroyed
D5		# offset	# offset	# offset/#0
D6	<logical-addr>	<phys-addr>	N/C	N/C
D7		N/C	N/C	N/C
A0	<TST-addr>	N/C	N/C	N/C
A1		N/C	N/C	N/C
A2		N/C	N/C	N/C
A3		N/C	N/C	N/C
A4		N/C	N/C	N/C
A5		N/C	N/C	N/C
A6		N/C	N/C	N/C
A7		N/C	N/C	N/C

NAME: PAGEALOC SBPAGAL	FUNCTION: Allocate physical memory pages.			PARA. B.4.9
	ENTRY	EXIT		
	CALL	RETURN 1	RETURN 2	RETURN 3
D0	#4	N/C	destroyed	
D1		# partition	destroyed	
D2		# page-count	destroyed	
D3		N/C	N/C	
D4		N/C	N/C	
D5		N/C	N/C	
D6		N/C	N/C	
D7		N/C	N/C	
A0	[<description>]	<phys-addr>	destroyed	
A1	[<phys-addr>]	destroyed	destroyed	
A2		destroyed	destroyed	
A3		N/C	N/C	
A4		N/C	N/C	
A5		N/C	N/C	
A6		N/C	N/C	
A7		N/C	N/C	

NAME: PAGEFREE SBPGFR	FUNCTION: Deallocate physical memory pages.			PARA. B.4.10
	ENTRY	EXIT		
	CALL	RETURN 1	RETURN 2	RETURN 3
D0	#5	destroyed	destroyed	
D1	# page-count	destroyed	destroyed	
D2		destroyed	destroyed	
D3		N/C	N/C	
D4		N/C	N/C	
D5		N/C	N/C	
D6		N/C	N/C	
D7		N/C	N/C	
A0	<address>	destroyed	destroyed	
A1		destroyed	destroyed	
A2		destroyed	destroyed	
A3		N/C	N/C	
A4		N/C	N/C	
A5		N/C	N/C	
A6		N/C	N/C	
A7		N/C	N/C	

B

NAME: PAUSE SBPAUSE	FUNCTION: Wait for external event.			PARA. B.4.11
	ENTRY	EXIT		
	CALL	RETURN 1	RETURN 2	RETURN 3
D0	#14	destroyed		
D1		N/C		
D2		N/C		
D3		N/C		
D4		N/C		
D5		N/C		
D6		N/C		
D7		N/C		
A0		N/C		
A1		N/C		
A2		N/C		
A3		N/C		
A4		N/C		
A5		N/C		
A6	<TCB-addr>	N/C		
A7		N/C		

NAME: PSEM4 SBP	FUNCTION: Request access to exclusive resource.			PARA. B.4.12
	ENTRY	EXIT		
	CALL	RETURN 1	RETURN 2	RETURN 3
D0	#1	destroyed		
D1		N/C		
D2		N/C		
D3		N/C		
D4		N/C		
D5		N/C		
D6		N/C		
D7		N/C		
A0	<semaphore>	N/C		
A1		N/C		
A2		N/C		
A3		N/C		
A4		N/C		
A5		N/C		
A6	<TCB-addr>	N/C		
A7		N/C		

NAME: RDTIMER SBRDTIM	FUNCTION: Read time of day.			PARA. B.4.13
	ENTRY	EXIT		
	CALL	RETURN 1	RETURN 2	RETURN 3
D0	#28	# TIMER		
D1		# millisecs		
D2		N/C		
D3		N/C		
D4		N/C		
D5		N/C		
D6		N/C		
D7		N/C		
A0		N/C		
A1		N/C		
A2		N/C		
A3		N/C		
A4		N/C		
A5		N/C		
A6		N/C		
A7		N/C		

NAME: READY SBREADY	FUNCTION: Move task to READY state.			PARA. B.4.14
	ENTRY	EXIT		
	CALL	RETURN 1	RETURN 2	RETURN 3
D0	#3	destroyed		
D1		N/C		
D2		N/C		
D3		N/C		
D4		N/C		
D5		N/C		
D6		N/C		
D7		N/C		
A0	<TCB-addr>	N/C		
A1		N/C		
A2		N/C		
A3		N/C		
A4		N/C		
A5		N/C		
A6		N/C		
A7		N/C		

NAME: VSEM4 SBV	FUNCTION: Release exclusive resource.			PARA. B.4.12
	ENTRY	EXIT		
	CALL	RETURN 1	RETURN 2	RETURN 3
D0	#2	destroyed		
D1		N/C		
D2		N/C		
D3		N/C		
D4		N/C		
D5		N/C		
D6		N/C		
D7		N/C		
A0	<semaphore>	N/C		
A1		N/C		
A2		N/C		
A3		N/C		
A4		N/C		
A5		N/C		
A6	<TCB-addr>	N/C		
A7		N/C		

NAME: WAKEUP SBWAKEUP	FUNCTION: To wake up task in current or next wait state.			PARA. B.4.15
	ENTRY	EXIT		
	CALL	RETURN 1	RETURN 2	RETURN 3
D0	#22	destroyed		
D1		N/C		
D2		N/C		
D3		N/C		
D4		N/C		
D5		N/C		
D6		N/C		
D7		N/C		
A0	<TCB-addr>	N/C		
A1		N/C		
A2		N/C		
A3		N/C		
A4		N/C		
A5		N/C		
A6		N/C		
A7		N/C		

APPENDIX C

VERSAdos TRAP #n MAP

C.1 INTRODUCTION

This appendix provides a map of the services provided by the TRAP functions.

C.2 TRAP #n MAP

TRAP NUMBER	DESCRIPTION
0	Services provided by RMS68K for PROCESSES.
1	Services provided by RMS68K for TASKS.
2	VERSAdos: Input/Output Services (reading and writing only).
3	VERSAdos: File Handling Services (open and close operations).
4	VERSAdos: LDR program loader.
5 - 15	User definable.

If an operating system other than VERSAdos is operating under the RMS68K Executive, TRAPs #2 through #4 are available for redefinition according to conventions defined by the alternate system.

THIS PAGE INTENTIONALLY LEFT BLANK.

APPENDIX D

CDB STRUCTURE

D.1 INTRODUCTION

This appendix describes the structure of Channel Data Blocks (CDBs) and Channel Control Blocks (CCBs).

D.2 CDB STRUCTURE

The structure of a CDB is provided in the following paragraphs. Included are the macro used to define CDBs and descriptions of the data block fields.

D.2.1 CDB Field Format

The file MACRO.DCB.SI contains a macro for defining CDBs. The format of this macro is as follows:

```
         * Macro to define a Channel Data Block (CDB)
         * Used by IOI to allocate channels
CDBLN  EQU $2E   Length of the CDB data structure.
CDB MACRO
 SECTION 1       === CDB SECTION ===
 DC.L *+CDBLN    Pointer to next CDB in list.
 DC.W \1         Options for the ALLOCATE command.
 DC.L \2         Channel mnemonic.
 DC.B \3         Channel type.
 DC.B \4         Masked interrupt maximum instruction count.
 DC.L \5         Physical address of driver.
 DC.L \6         Supervisor channel's mnemonic (only if bit 3 of options set).
 DC.L \7         Physical address of device in memory mapped I/O space.
 DC.W \8         # of bytes device occupies in memory mapped I/O space.
 DC.B \9         Vector number.
 DC.B \A         Polling priority.
 DC.B \B         Software priority.
 DC.B \C         Segment count (number of polling entries).
 DC.W \D         Polling byte offset. -- [#1] --
 DC.B \E         Polling mask.
 DC.B \F         Polling test value.
 DC.W \G         Offset from physical device address for reset.
 DC.B \H         Value for reset.
 DC.B 0          Reserved.
 DC.W \I         Polling byte offset. -- [#2] --
 DC.B \J         Polling mask.
 DC.B \K         Polling test value.
 DC.W \L         Offset from physical device address for reset.
 DC.B \M         Value of reset.
 DC.B 0          Reserved.
 ENDM
```

D.2.2 Data Block Field Descriptions

Field descriptions for CDBs are as follows:

Options -

Bit 0 = 0	Attach requests from any task are honored.
= 1	Attach requests from system tasks only are honored.
Bit 1	Reserved.
Bit 2	Exclusive vectoring.
Bit 3	Channel is to be subordinate to Supervisor Channel Mnemonic field.
Bit 4	Channel is to be a supervisor.

Channel Mnemonic -

Uniquely identifies the channel. It must be nonzero and it must be a distinct mnemonic from any other channel or value name in the system.

Channel Type -

A code which indicates the physical type of the channel. Interpretation of this code will cause the CMR handler to handle the INITIATE I/O command in one of four ways. Channels with code $01 through $0F will pass the request to the I/O handler with no parameter block checking (nonstandard CMR channels). Channels with code $10 through $7F will perform parameter block checks (standard CMR channels and serial port channels). Channels with code $FF will only notify a task of an interrupt occurring. Channels with code $80 through $8F will perform parameter block checks (shared-access channels).

Physical Address of Driver -

This field points to an I/O handler. The structure of the I/O handler must have a service vector table that defines:

- The interrupt service entry point
- The INITIATE I/O command service entry point
- The initialization entry point
- A longword for future use

These are 4-byte fields of absolute addresses. This field is not required for channel type $FF, which assumes the interrupt is cleared when the CMR handler polls the device and finds it activated.

Base Address of Memory Mapped I/O Space -

Physical address of a readable byte in memory of the memory mapped I/O space for this channel, usually a status register on the device.

Length of Memory Mapped I/O Space -

Number of bytes in the memory mapped I/O space for this channel.

Hardware Vector Number -

Indicates the associated hardware vector. It must be an auto vector (values 25 through 31) or a user vector (64 through 255).

Hardware Priority Level -

Indicates the hardware interrupt level associated with the channel. It must be a value in the range of 1 through 7, inclusive.

Software Priority Number -

Indicates the position of the CDB within the polling chain. A higher value of the software priority level will result in faster service to the channel when it interrupts.

Number of Polling Table Entries -

Indicates the number of 8-byte polling table entries which follow. This number can be in the range of 1 through 4, inclusive. For each type of interrupt associated with a channel, one 8-byte table entry is required to describe the details of that interrupt. When an interrupt occurs, the following algorithm is used to determine if this particular channel caused an interrupt.

The polling byte defined by the polling byte offset is read. If the polling test value is zero, the polling byte is complemented; otherwise, it is left unchanged. The resulting polling byte is then ANDed with the polling mask. If this result is nonzero, it is assumed that this channel caused the interrupt.

These entries are used only for interrupt-only channels and for channels which are nonstandard.

Polling Byte Offset -

The zero-relative offset from the base of memory mapped I/O space for this channel where the polling byte resides. Refer to the polling algorithm described in "Number of Polling Table Entries".

Polling Mask -

Used in the polling algorithm to determine if this channel caused an interrupt. Refer to the polling algorithm described in "Number of Polling Table Entries".

Polling Test Value -

Used in the polling algorithm to determine the polling byte value. Refer to the polling algorithm described in "Number of Polling Table Entries".

Reset Byte Offset -

The zero-relative offset from the base of memory mapped I/O space for this channel where the reset byte resides.

Reset Value -

Used by the interrupt service to clear random or unexpected interrupts on a channel. It is critical that the reset byte offset and reset value be defined correctly to prevent infinite loops in the polling routine caused by interrupts that cannot be cleared.

D.2.3 CDB Macro Example

An example of the CDB macro is illustrated in Figure 1.

D.3 CCB STRUCTURE

Some of the universally defined fields in a CCB are used primarily by the CMR handler, and others are important to the I/O device driver that manages the channel. The structure of a CCB is diagrammed as follows:

The following fields in a CCB are set at ALLOCATE time.

```
        |----------------+----------------+----------------+----------------|
   $00  |  "!CCB" (Eye catcher to aid in dump analysis)                     |
        |----------------+----------------+----------------+----------------|
   $04  |  CCBALL (Link to the next CCB in the list of all CCBs)            |
        |----------------+----------------+----------------+----------------|
   $08  |  CCBPOLL (Link to next CCB in polling list for interrupt vector)  |
        |----------------+----------------+----------------+----------------|
```

The next field of a CCB is set at ATTACH time.

```
        |----------------+----------------+----------------+----------------|
   $0C  |  CCBTLNK (Link to the next CCB attached to the same task)         |
        |----------------+----------------+----------------+----------------|
                        ***  CHANNEL DATA BLOCK  ***
0 00000168                     CDB      $0009.CNAME.CHANTYP2.254,MPSC,SNAME,DEVADDR1+PORTB.1,IOCVECT,IOCLVL,
0 00000168              &               $30.0.0.0 0 0.0,0,0,0,0,0
           000000001           SECTION  1                     === CDB SECTION ===
1 0000005C 00000008A           DC.L     *+CDBLN               Pointer to next CDB in list.
1 00000060 0009                DC.W     $0009                 Options for the ALLOCATE command.
1 00000062 434D5031            DC.L     CNAME                 Chonnel mnemonic.
1 00000066 65                  DC.B     CHANTYP2              Channel type.
1 00000067 FE                  DC.B     254                   Masked interrupt maximum instruction count.
1 00000068 00009B00            DC.L     MPSC                  Physicol address of driver.
1 0000006C 53505230            DC.L     SNAME                 Supervisor channel's mnemonic (anly if bit 3 of aptians set).
1 00000070 00FFE1CF            DC.L     DEVADDR1+PORTB        Physicol oddress of device in memory-mapped I/O space.
1 00000074 0001                DC.W     1                     of bytes device accupies in memory-mopped I/O space.
1 00000076 52                  DC.B     IOCVECT               Vector number.
1 00000077 04                  DC.B     IOCLVL                Polling priority.
1 00000078 30                  DC.B     $30                   Software priority.
1 00000079 00                  DC.B     0                     Segment count. (Number of polling entries)
1 0000007A 0000                DC.W     0                     Polling byte offset. — 1] —
1 0000007C 00                  DC.B     0                     Polling mosk.
1 0000007D 00                  DC.B     0                     Polling test value.
1 0000007E 0000                DC.W     0                     Offset from physical device address for reset.
1 00000080 00                  DC.B     0                     Volue for reset.
1 00000081 00                  DC.B     0                     Reserved.
1 00000082 0000                DC.W     0                     Polling byte offset.  —2] —
1 00000084 00                  DC.B     0                     Polling mask.
1 00000085 00                  DC.B     0                     Polling test volue.
1 00000086 0000                DC.W     0                     Offset from physical device address for reset.
1 00000088 00                  DC.B     0                     Volue for reset.
1 00000089 00                  DC.B     0                     Reserved.
```

Figure 1. CDB Macro Example

The following fields in a CCB are set at ALLOCATE time.

```
     |----------------+----------------+----------------+----------------|
$10  |  CCBSUB (Link to CCB of next subordinate channel if one exists)   |
     |----------------+----------------+----------------+----------------|
$14  |  CCBMNEM (Unique channel ID mnemonic for this channel)            |
     |----------------+----------------+----------------+----------------|
$18  |  CCBTYPE       | <-- (Channel type; $10 - $5F for standard chan.
     |----------------|                    $60 - $7F for serial port chan.)
$19  |  CCBCMCT       | <-- (For CHPI, Channel Program Interpreter)
     |----------------+----------------+----------------+----------------|
$1A  |  CCBUSER (Physical address of driver's command service routine)   |
     |----------------+----------------+----------------+----------------|
$1E  |  CCBRTNI (Physical address of driver's interrupt service routine) |
     |----------------+----------------+----------------+----------------|
$22  |  CCBCHB (Physical address of channel's memory mapped I/O space)   |
     |----------------+----------------+----------------+----------------|
$26  |  CCBMME                         |  <-- (Length of I/O memory space)
     |----------------+----------------|
$28  |  CCBVECT       |  <-- (Vector number of interrupt for this channel)
     |----------------|
$29  |  CCBPPRIO      |  <-- (Hardware interrupt priority level; 1-7)
     |----------------|
$2A  |  CCBSPRIO      |  <-- (Software interrupt polling priority level)
     |----------------|
$2B  |  CCBPECT       |  <-- (Number of polling table entries for a non-
     |----------------|       standard or interrupt-only channel)
```

The following fields in a CCB are set at ATTACH time.

```
     |----------------+----------------+----------------+----------------|
$2C  |  CCBRQST (Taskname and session number of the task that is         |
     |--          attached to the channel;  I/O events should be       --|
     |            queued to this task, which is normally IOS)            |
     |----------------+----------------+----------------+----------------|
$34  |  CCBRQSTA (Physical address of the above task's TCB)             |
     |----------------+----------------+----------------+----------------|
$38  |  CCBKEY        |  <-- (Key supplied by task attached to channel)
     |----------------|
$39  |  CCBRTN        |  <-- (Return option; specified by attached task)
     |----------------|
$3A  |  CCBASQL       |  <-- (ASQ entry length for attached task)
     |----------------|
$3B  |  Unused        |  <-- (Necessary for word alignment)
     |----------------+----------------+----------------+----------------|
$3C  |  CCBSVVC (Logical address of service routine for channel events)  |
     |----------------+----------------+----------------+----------------|
```

The following fields in a CCB are set at ALLOCATE time.

```
     |---------------+---------------|
$40  |  CCBSR (SR value to enable int) |  <-- (Enables channel interrupts)
     |---------------+---------------|
$42  |  CCBISR (SR value to mask int)  |  <-- (Masks channel interrupts)
     |---------------+---------------+---------------+---------------|
$44  |  CCBVADR (Physical address of this channel's interrupt vector)  |
     |---------------+---------------+---------------+---------------|
$48  |  CCBSTR (Flag bits used by CMR) |  <-- (Dynamically updated by CMR)
     |---------------+---------------|
$4A  |  CCBJSR (JSR instruction; sends |
     |--         control to the CMR  --|
     |           handler when an       |
     |--         interrupt for this  --|
     |           channel is received)  |
     |---------------+---------------|
```

The following fields in a CCB are set for each command to INITIATE I/O.

```
     |---------------+---------------+---------------+---------------|
$50  |  CCBTASKA (Physical address of the TCB of task that owns IOCB)  |
     |---------------+---------------+---------------+---------------|
$54  |  CCBTASK (Taskname and session number of the task that owns the |
     |--          IOCB for the latest I/O request; this field and the --|
     |            field above are valid until a new I/O request occurs) |
     |---------------+---------------+---------------+---------------|
```

The following CCB fields are reserved for future device-independent use.

```
     |---------------+---------------+---------------+---------------|
$5C  |          (Reserved for future device-independent use)           |
     :                               .                                 :
     :                               .                                 :
     :                               .                                 :
$6C  |                                                                 |
     |---------------+---------------+---------------+---------------|
```

The remaining fields in a CCB are reserved for device-dependent variables.

```
     |---------------+---------------+---------------+---------------|
$70  |                  (Device-dependent variables)                   |
     :                               .                                 :
     :                               .                                 :
     :                               .                                 :
     |                                                                 |
$FF  |---------------+---------------+---------------+---------------|
```

The individual universally defined fields of a CCB are specified by the OFFSET block that follows, and the fields are described in detail as they are defined.

	OFFSET	0	This OFFSET block defines the structure for a CCB.
	DS.B	4	The CMR handler sets this field at ALLOCATE time to contain "!CCB" as an eye catcher to aid in the analysis of system dumps.
CCBALL	DS.B	4	This field contains a link to the next CCB in the list of all CCBs. All CCBs in the entire system are linked together in the order in which they are allocated, and a zero link value marks the end of the list. A zero link value is universally used to mark the end of any CCB list. All the CCBs in the entire system must be linked together into a list so that the CMR handler can verify at ALLOCATE time that the channel ID mnemonic for the channel that is being allocated is unique. The CMR handler also uses the list of all CCBs to find the CCB that corresponds to the channel ID mnemonic that is given in a CMR parameter block with an I/O request.
CCBPOLL	DS.B	4	This field contains a link to the next CCB in the polling list for the channel's interrupt vector. Several I/O devices typically share a single interrupt and, thus, share the same interrupt vector. All of the CCBs for the devices on a single interrupt vector are linked together into a polling list, and a zero link value marks the end of the polling list. When an I/O interrupt occurs, the CMR handler receives control. The CMR handler then goes through the interrupt vector's polling list to give the device driver for each channel in the polling list a chance to answer the interrupt. This polling process terminates when some device driver claims the interrupt as its own. CCBs near the front of a polling list have higher software priority than the CCBs near the back of the polling list because drivers for CCBs at the front of the polling list are polled first.
CCBTLNK	DS.B	4	This field contains a link to the next CCB that is attached to the same task. In actual practice, all channels are attached to IOS, but a channel could be attached to any task. If a task were to be aborted, the system might wish to detach all channels that were attached to that task.

9CBSUB	DS.B	4	This field contains a link to the next CCB in the list of subordinate CCBs. Subordinate CCBs arise when multiple CCBs are required to provide service for a single hardware device. In this case, there are one supervisor CCB and two or more subordinate CCBs for the hardware device, and the CCBSUB field of the supervisor CCB is used to link the supervisor CCB to the list of subordinate CCBs for the device. The CCBSUB field of each subordinate CCB contains a link to the next subordinate CCB in the list, and a zero link value marks the end of the list. The interrupt-handling routine of the driver for the supervisor CCB receives control from the CMR handler when the hardware device generates an interrupt. The supervisor interrupt handler determines what particular service is required, and then the supervisor interrupt handler passes control to the interrupt handler of the subordinate driver that provides the required service. Thus, the interrupt handler for a subordinate driver receives control only after the interrupt handler for the supervisor driver has determined that the subordinate interrupt handler should receive control, and subordinate CCBs do not appear directly in any polling lists.

The fields from CCBMNEM through CCBPECT must always be kept together in consecutive memory locations because there is code in the system that accesses all of these fields with a single MOVEM.L instruction.

CCBMNEM	DS.B	4	This field of a CCB contains the channel ID mnemonic. The channel ID mnemonic is established at ALLOCATE time (by IOI in practice), and the channel ID mnemonic then serves as a unique identifier for the channel. The channel ID mnemonic must be nonzero, and it must not be duplicated by any other channel ID mnemonic.
CCBTYPE	DS.B	1	This field is set up at ALLOCATE time to specify the channel type. General classifications of type codes are as follows: \$01 - \$0F Nonstandard CMR channels. \$10 - \$5F Other standard channels. \$60 - \$7F Serial port channels. \$80 - \$8F Shared-access channels. \$FF Interrupt-only channels.
CCBCMCT	DS.B	1	This field is used by the Channel Program Interpreter (CHPI), and is retained only for purposes of backward compatibility.

CCBUSER	DS.B	4	The CMR handler sets up this field at ALLOCATE time to contain the physical address of the Driver's Command Service Routine (DCSR). The CMR handler obtains the address of the DCSR from a vector table that is located at the beginning of the driver. The CMR handler later uses the CCBUSER field of the CCB when it needs to invoke the DCSR to respond to a command to initiate an I/O transaction.
CCBRTNI	DS.B	4	The CMR handler sets up this field at ALLOCATE time to contain the physical address of the Driver's Interrupt Service Routine (DISR). The CMR handler obtains the address of the DISR from a vector table that is located at the beginning of the driver. The CMR handler later uses the CCBRTNI field of the CCB when it needs to invoke the DISR to respond to an interrupt.
CCBCHB	DS.B	4	The CMR handler sets up this field at ALLOCATE time to contain the physical memory address of some byte (usually the first byte) of the channel's memory mapped I/O area. All communication with the I/O device(s) on the channel is done through this memory mapped I/O area, and the device driver uses the CCBCHB field of the CCB to find out where the memory mapped I/O area is located. The CMR handler uses this address to do a read from the channel to determine if it is physically attached. Therefore, this must be a readable location.
CCBMME	DS.B	2	The CMR handler sets up this field at ALLOCATE time to tell how many bytes of the MC68000 memory space are dedicated to the channel's memory mapped I/O space. Although some bytes within the memory mapped I/O space might not be used, they must still be included in the byte count (e.g., M6800 peripherals occurring only on odd addresses). The CCBMME value is utilized by a system-level initialization routine. That routine reads one or more bytes (as indicated by the CCBMME value) from the memory mapped I/O area to verify the presence or discover the absence of the I/O channel in question. A bus error indicates that the I/O channel in question is missing. The lack of a bus error indicates that the channel exists.

CCBVECT	DS.B	1	The CMR handler sets up this field at ALLOCATE time to contain the vector number of the interrupt vector that corresponds to this channel. The vector number is in the range 25 through 31 if the channel uses an auto-vector interrupt, and in the range 64 through 255 if the channel hardware provides its own interrupt vector number at interrupt time. The value in this field must agree with the actual configuration of the system.
CCBPPRIO	DS.B	1	The CMR handler sets up this field at ALLOCATE time with the hardware interrupt priority level for the channel's interrupt. The MC68000 recognizes interrupt priority values in the range 1-7, so the value in this field is in that range. The highest possible priority is 7; the lowest is 1.
CCBSPRIO	DS.B	1	The CMR handler sets up this field at ALLOCATE time to contain the software interrupt priority value. The CMR handler uses the unsigned software interrupt priority value as a key for maintaining a sorted polling list of the CCBs that share a common interrupt, and CCBs with higher software interrupt priority values end up closer to the front of the polling list. The software interrupt priority value is in the range 0 through 255.
CCBPECT	DS.B	1	The CMR handler sets up this field at ALLOCATE time to tell how many entries there are in the channel's polling table. This value is not meaningful for standard channels; it is useful for nonstandard channels and interrupt-only channels.
CCBRQST	DS.B	4*2	The CMR handler sets up this field at ATTACH time to contain task ID and session number of the task attached to the channel. Channel I/O events should be queued to this task, which is normally IOS.
CCBRQSTA	DS.B	4	The CMR handler sets up this field at ATTACH time to contain the physical address of the Task Control Block (TCB) for the task that is attached to the channel. Channel I/O events should be queued to this task, which is normally IOS.

CCBKEY	DS.B	1	The CMR handler sets up this field at ATTACH time to contain a key that is supplied by the task (normally IOS) attached to the channel. The device driver includes this key value in all I/O events that it queues to the attached task, so the attached task can use this key value to identify the particular channel associated with any given I/O event. Therefore, the CCBKEY value serves as a channel identifier that is unique among the channels that are attached to a particular task.
CCBRTN	DS.B	1	The CMR handler sets up this field at ATTACH time to indicate how the device driver should notify the attached task of the completion of an I/O transaction. The values in this field are interpreted as follows. 0: Put the completion status into a buffer and issue a wakeup to the attached task. 1: Put the completion status into an event, and queue the event to the attached task. The attached task will receive the event in its Asynchronous Service Queue (ASQ). The CCBRTN field of the CCB is not recognized by VERSAdos device drivers. VERSAdos device drivers assume that the value of the CCBRTN field is 1.
CCBASQL	DS.B	1	The CMR handler sets up this field at ATTACH time to tell how many bytes the attached task can accept in an event entry of the task's ASQ. The device driver must limit the length of event messages to that specified in this field of the CCB. This field is meaningless unless the value of the CCBRTN field is 1.
	DS.B	1	This field of a CCB is not used. It is reserved because the next field of a CCB must be aligned on a word boundary.
CCBSVVC	DS.B	4	The CMR handler sets up this field at ATTACH time to contain the logical address of the routine (in the attached task) that should be invoked to respond to an I/O event from this channel. A task's Asynchronous Service Routine (ASR) normally responds to all events, but a task can specify a special routine that is to be invoked in response to events from a particular channel.

			If the attached task does not specify a special routine to be invoked in response to I/O events from this channel, the CMR handler simply clears the CCBSVVC field to indicate that the task's ASR will respond to I/O events. The CCBSVVC field of the CCB is meaningless unless the value of the CCBRTN field is 1.
CCBSR	DS.B	2	The CMR handler sets up this field at ALLOCATE time to contain a Status Register (SR) value that will enable interrupts in supervisor state at the hardware priority level of the channel's interrupt. Therefore, the value of the CCBSR field is $2x00, where x equals one less than the value of the CCBPPRIO field. Bit 13 of the CCBSR field is set because I/O device drivers always run in the supervisor state. A device driver must run in the supervisor state so it can change the interrupt level in the SR. When a device driver's interrupt handling routine receives control, interrupts at the level of the channel's interrupt are masked by the SR. If the interrupt belongs to the driver, the driver's interrupt handling routine should quickly mask or clear the interrupt at the device level and then set the SR equal to the CCBSR value as soon as possible to allow interrupts at the channel's priority level. This procedure is necessary because other channels typically run at the same interrupt level, and the interrupts from these other channels of equal priority should not be masked any longer than is absolutely necessary. Interrupts at lower priority levels can remain masked while the interrupt handling routine for this channel finishes its work.
CCBISR	DS.B	2	The CMR handler sets up this field at ALLOCATE time to contain an SR value that will mask interrupts in supervisor state at the hardware priority level of the channel's interrupt. Therefore, the value of the CCBISR field is $2y00, where y equals the value of the CCBPPRIO field. Bit 13 of the CCBISR field is set because device drivers always run in the supervisor state. A driver must run in the supervisor state so that it can change the interrupt level in the SR.

When a device driver needs to mask all interrupts at the level of the channel's interrupt, it can set the SR equal to the value of the CCBISR field. A driver should mask interrupts at its interrupt level only briefly. Other channels typically run at the same interrupt level, and interrupts from these other channels of equal priority should not be masked any longer than is absolutely necessary.

When a DCSR is invoked, all interrupts at all levels are enabled. The DCSR should keep the interrupt priority level set at zero as much as possible.

CCBVADR	DS.B	4	The CMR handler sets up this field at ALLOCATE time with the physical address of the interrupt vector for this channel's interrupt. A device driver can use this address if it must simulate an interrupt. Sometimes a simulated interrupt is helpful when a driver is trying to start a device.
CCBSTR	DS.B	2	This field contains some flag bits used by the CMR handler.
CCBFGATH	EQU	15	
CCBJSR	DS.B	6	The CMR handler sets up this field at ALLOCATE time to contain a JSR instruction that addresses the CMR interrupt handling routine. The actual interrupt vector for a channel points to the CCBJSR field of the first CCB in the interrupt vector's polling list, and the JSR instruction there transfers control to the CMR handler when an interrupt for a channel in the polling list occurs. The PC value that the JSR instruction pushes onto the stack allows the CMR handler to locate the first CCB in the interrupt vector's polling list. Therefore, the CMR handler can start polling the device drivers from the polling list to find the one that controls the interrupting device.
CCBIOH	EQU		The following fields of a CCB are set by the CMR handler when a command to initiate an I/O transaction is received, so this part of a CCB changes every time a task requests an I/O transaction.
CCBTASKA	DS.B	4	This field contains the physical address of the TCB for the task that owns the IOCB for the last I/O request that was issued for this channel.
CCBTASK	DS.B	4*2	This field contains the task ID and the task session number of the task that owns the IOCB for the last I/O request that was issued for this channel.

CCBRESV1 CCBRESV2 CCBRESV3 CCBRESV4 CCBRESV5	DS.L DS.L DS.L DS.L DS.L	1 1 1 1 1	These universally defined CCB fields are reserved for future use. By using these fields for future universally defined CCB variables, the universally defined CCB structure can be extended with no need to reassemble all the existing programs that use the CCB.
CCBDDP	EQU		The remaining part of a CCB is available for use by the channel's driver for device-dependent variables.

THIS PAGE INTENTIONALLY LEFT BLANK.

APPENDIX E

DCB STRUCTURE

E.1 INTRODUCTION

This appendix defines the macros used to define Device Control Blocks (DCBs). These macros are located in the VERSAdos file MACRO.DCB.SI, which is released by Motorola under user number 9998. Users may move this file into user number 9100 with the COPYSGEN chainfile. Refer to the System Generation Facility User's Manual for details about COPYSGEN, and to Chapters 5 and 6 of this guide for the correct placement of COPYSGEN in the driver addition procedure. DCB examples are provided as sample data structures for driver writers.

E.2 MACROS for DCBs

```
*****************************************************************************
* MACRO.DCB.SI -- DEFINES BASIC DCB AND CDB MACROS
*   DCB = Device Control Block:  contains information about device, including
*         default configuration; used heavily by FMS, FHS, and IOS.
*****************************************************************************
*  DIPDCB macro defines device-independent portion of a DCB.  Used by the
*  other device-dependent macros DSKDCB, CRTDCB, PRTDCB.
DIPDCB MACRO
 DC.L    *+\1          Address of next DCB in linked list.
 DC.L    \2            ASCII identification for this DCB.
 DC.L    0             Address of DCQ entry.
 DC.L    \3            Name of task making the request.
 DC.L    \4            Session of task making the request.
 DC.L    0             Address of LUT.
 DC.L    \5            Device attributes associated with DCB.  BKM 7/12/83
 DC.W    0             Write/read protect codes.
 DC.W    0             "Device in use" flag.
 DC.L    0             Write/read counts.
 DC.B    \6            Device flag (device code).
 DC.B    \7            Device flag (device status).
 DC.L    \8            Channel identification.
 DC.B    \9            Device number associated with the channel.
 DC.B    0             Task priority.
 DC.L    0             Current record #.
 DS.B    IOSBLN        Storage area for the IOCB being processed.
 DC.L    0             Logical address of IOCB in user's address space.
 DC.B    0             Configuration coordination flag (0 --> at defaults).
 DC.B    0             Break count.
 DC.L    0             Address of break service LUT.
 DC.L    0             Break service address.  -----------------
 DC.L    0             Event claimer -- taskname.               BKM 7/12/83
 DC.L    0             Event claimer -- session number.         BKM 7/12/83
 DC.L    \A            Address of supervisor DCB or session     BKM 7/12/83
*                      number if this is a supervisor DCB.      BKM 7/12/83
 DC.L    0             Supervisor/subordinate DCB open count.   BKM 7/12/83
 DC.L    0,0,0,0       Device independent/dependent buffer zone.
 ENDM
```

```
*  Macro to define DCB for a disk.
*
DSKDCB MACRO
 SECTION 0       === DCB SECTION ===
 DIPDCB DDCBLN,\1,\2,\3,\4,\5,\6,\7,\8,0
 DC.B    0,0,0,0 Space for status fields.
 DC.W    \9      Attributes mask.
 DC.W    \A      Parameters mask.
 DC.W    \B      Attributes word.
 DC.W    \C      # of bytes/sector.
 DC.L    \D      Total # of sectors -- returned information.
 DC.L    \E      Write time-out.
 DC.L    \F      Read time-out.
 DC.B    \G      # of sectors/track.
 DC.B    \H      # of heads.
 DC.W    \I      # of cylinders on media.
 DC.B    \J      Interleave factor.
 DC.B    \K      Spiral offset (in sectors).
 DC.W    \L      Physical sector size of media.
 DC.W    \M      Starting head number on drive.
 DC.W    \N      Number of cylinders on drive.
 DC.W    \O      Precompensation cylinder number.
 DC.B    \P      Physical sectors per track on drive.
 DC.B    \Q      Stepping rate.
 DC.W    \R      Reduced write current cylinder number.
 DC.W    \S      ECC data burst length.
 DC.B    0,0     2 bytes reserved as offset to another parameter block.
 ENDM
*
*  Macro to define DCB for a terminal.
CRTDCB MACRO
 DIPDCB CDCBLN,\1,\2,\3,\4,\5,\6,\7,\8
 DC.B    0,0,0,0 Space for status fields.
 DC.W    \9      Attributes mask.
 DC.W    \A      Parameters mask.
 DC.W    \B      Attributes word.
 DC.W    \C      # of characters/line.
 DC.L    \D      # of lines/page.
 DC.L    \E      Write time-out.
 DC.L    \F      Read time-out.
 DC.B    \G      XOFF character.
 DC.B    \H      XON character.
 DC.B    \I      BREAK EQUIVALENT character.
 DC.B    \J      DISCARD OUTPUT character.
 DC.B    \K      REPRINT LINE character.
 DC.B    \L      CANCEL LINE character.
 DC.L    \M      Read terminators.
 DC.L    \N      End-of-line string.
 DC.B    \O      BAUD rate code.
 DC.B    \P      NULL padding.
 DC.B    \Q      Terminator class.
 DC.B    \R      Terminal type (0=EXORterm 155).
 DC.B    0,0,0,0,0,0 Reserved.
 ENDM
```

```
*
*  Macro to define DCB for a printer.
*
PRTDCB MACRO
 DIPDCB PDCBLN,\1,\2,\3,\4,\5,\6,\7,\8
 DC.B    0,0,0,0   Space for status fields.
 DC.W    \9        Attributes mask.
 DC.W    \A        Parameters mask.
 DC.W    \B        Attributes word.
 DC.W    \C        # of characters/line.
 DC.L    \D        # of lines/page.
 DC.L    \E        Write time-out.
 DC.L    0         Read time-out.
 DC.W    \F        Logical line length.
 DC.B    \G        End-of-line character.
 DCB.B   15,0      Reserved space (15 bytes).
 ENDM
```

E.3 DCB MACRO EXAMPLES

E.3.1 A Terminal DCB Example

The following input parameters:

```
         /1 CN10       /B TCP$ATW         /L $18
         /2 IOSID      /C 80              /M $0DDE0000
         /3 IOSESS     /D 24              /N $0D0A0000
         /4 $133       /E 120000          /O $03
         /5 30         /F 900000          /P 0
         /6 1          /G $17             /Q $00
         /7 COM1       /H 0               /R $00
         /8 0          /I $03
         /9 $47C       /J $0F
         /A 1801       /K $13
```

```
CRTDCB 'CN10',IOSID,IOSESS,$133,30,1,'COM1',0,$047C,$1801,TCP$ATW,80,24,
       120000,900000,$17,0,$03,$0F,$13,$18,$0DDE0000,$0D0A0000,$03,0,$00,$00
```

were used to create the following terminal DCB:

```
DIPDCB   CDCBLN,'CN10',IOSID,IOSESS,$133,30,1,'COM1',0
DC.L     *+CDCBLN       Address of next DCB in linked list.
DC.L     'CN10'         ASCII identification for this DCB.
DC.L     0              Address of DCQ entry.
DC.L     IOSID          Name of task making the request.
DC.L     IOSESS         Session of task making the request.
DC.L     0              Address of LUT.
DC.W     $133           Attributes of device associated with this DCB.
DC.W     0              Write/read protect codes.
DC.W     0              "Device in use" flag.
DC.L     0              Write/read counts.
DC.B     30             Device flag (device code).
DC.B     1              Device flag (device status).
DC.L     ’COM1’          Channel identification.
DC.B.    0               Device number associated with the channel.
DC.B     0               Task priority.
DC.L     0               Current record number.
DC.B     IOSBLN          Storage area for the IOCB being processed.
DC.L     0               Logical address of IOCB in user’s address space.
DC.B     0               Configuration coordination flag (0 --> at defaults).
DC.B     0               Break count.
DC.L     0               Address of break service LUT.
DC.L     0               Break service address.
DC.B     0,0,0,0         Space for status fields.
DC.W     $047C           Attributes mask.
DC.W     $1801           Parameters mask.
DC.W     TCP$ATW         Attributes word.
DC.W     80              Number of characters/line.
DC.L     24              Number of lines/page.
DC.L     120000          Write time-out.
DC.L     900000          Read time-out.
DC.B     $17             XOFF character.
DC.B     0               XON character.


DC.B     $03             Break equivalent character.
DC.B     $0F             Discard output character.
DC.B     $13             Reprint line character.
DC.B     $18             Cancel line character.
DC.L     $0DDE0000       Read terminators.
DC.L     $0D0A0000       End-of-line string.
DC.B     $0E             Baud rate code.
DC.B     0               NULL padding.
DC.B     $00             Terminator class.
DC.B     $00             Terminal type (0=EXORterm 155).
DC.B     0,0,0,0,0,0     Reserved.
```

E.3.2 A Printer DCB Example

The following input parameters:

```
        /1  PRTDV           /B  PCP$ATW
        /2  IOSID           /C  132
        /3  IOSESS          /D  66
        /4  $632            /E  120000
        /5  95              /F  132
        /6  1               /G  $0A
        /7  CPRT
        /8  0
        /9  $0007
        /A  $0033
```

```
PRTDCB    PRTDV,IOSID,IOSESS,$632,95,1,’CPRT’,0,$0007,$0033,PCP$ATW,132,
          66,120000,132,$0A
```

were used to create the following printer DCB:

```
DIPDCB   PDCBLN,PRTDV,IOSID,IOSESS,$632,95,1,'CPRT',0
DC.L     *+PDCBLN         Address of next DCB in linked list.
DC.L     PRTDV            ASCII identification for this DCB.
DC.L     0                Address of DCQ entry.
DC.L     IOSID            Name of task making the request.
DC.L     IOSESS           Session of task making the request.
DC.L     0                Address of LUT.
DC.W     $632             Attributes of device associated with this DCB.
DC.W     0                Write/read protect codes.
DC.W     0                "Device in use" flag.
DC.B     0                Write/read counts.
DC.B     95               Device flag (device code).
DC.B     1                Device flag (device status).
DC.L     'CPRT'           Channel identification.
DC.B     0                Device number associated with the channel.
DC.B     0                Task priority.
DC.L     0                Current record number.
DS.B     IOSBLN           Storage area for the IOCB being processed.
DC.L     0                Logical address of IOCB in user's address space.
DC.B     0                Configuration coordination flag (0--> at defaults).
DC.B     0                Break count.
DC.L     0                Address of break service LUT.
DC.L     0                Break service address.
DC.B     0,0,0,0          Space for status fields.
DC.W     $0007            Attributes mask.
DC.W     $0033            Parameters mask.
DC.W     PCP$ATW          Attributes word.
DC.W     132              Number of characters/line.
DC.L     66               Number of lines/page.
DC.L     120000           Write time-out.
DC.L     0                Read time-out.
DC.W     132              Logical line length.
DC.B     $0A              End-of-line character.
DC.B     15,0             Reserved space (15 bytes).
```

E.3.3 A Disk DCB Example

The following input parameters:

```
/1  DSKNM       /B  DCP$ATW
/2  IOSID       /C  256
/3  IOSESS      /D  0
/4  $1F         /E  0
/5  40          /F  0
/6  4           /G  064
/7  CRD1        /H  0
/8  0           /I  0
/9  $FFFF       /J  0
/A  $FFF3       /K  0
```

```
DSKDCB   DSKNM,IOSID,IOSESS,$1F,40,4,’CRD1’,0,$FFFF,$FFF3,DCP$ATW,
         256,0,0,0,064,0,0,0,0
```

were used to created the following disk DCB:

```
DIPDCB   DDCBLN,DSKNM,IOSID,IOSESS,$1F,40,4,’CRD1’,0
DC.L     *+DDCBLN          Address of next DCB in linked list.
DC.L     DSKNM             ASCII identification for this DCB.
DC.L     0                 Address of DCQ entry.
DC.L     IOSID             Name of task making the request.
DC.L     IOSESS            Session of task making the request.
DC.L     0                 Address of LUT.
DC.W     $1F               Attributes of device associated with this DCB.
DC.W     0                 Write/read protect codes.
DC.W     0                 "Device in use" flag.
DC.L     0                 Write/read counts.
DC.B     40                Device flag (device code).
DC.B     4                 Device flag (device status).
DC.L     ’CRD1’            Channel identification.
DC.B     0                 Device number associated with the channel.
DC.B     0                 Task priority.
DC.L     0                 Current record number.
DS.B     IOSBLN            Storage area for the IOCB being processed.
DC.L     0                 Logical address of IOCB in user’s address space.
DC.B     0                 Configuration coordination flag (0 --> at defaults).
DC.B     0                 Break count.
DC.L     0                 Address of break service LUT.
DC.L     0                 Break service address.
DC.B     0,0,0,0           Space for status fields.
DC.W     $FFFF             Attributes mask.
DC.W     $FFF3             Parameters mask.
DC.W     DCP$ATW           Attributes word.
DC.W     256               Number of bytes/sector.
DC.L     0                 Total number of sectors -- returned information.
DC.L     0                 Write time-out.
DC.L     0                 Read time-out.
DC.B     064               Number of sectors/track.
DC.B     0                 Number of heads.
DC.W     0                 Number of tracks -- returned information.
DC.B     0                 Interleave factor.
DC.B     0                 Spiral offset (in sectors).
DC.L     0,0,0,0           16 bytes reserved for future use.
DC.W     0                 Polling byte offset. -- [#1] --
DC.B     $80               Polling mask.
DC.B     $80               Polling test value.
DC.W     0                 Offset from physical device address for reset.
DC.B     3                 Value for reset.
DC.B     0                 Reserved.
```

APPENDIX F

SAMPLE FILES CREATED BY THE DRIVER WRITER

F.1 INTRODUCTION

This appendix contains examples of the file types that driver writers must create or modify in order to add their drivers into the system. Examples of the following types are provided.

- Assembly chainfile
- Link file for SYSGEN
- Configuration files (DCBs and CDBs)
- Patch chainfile
- INCLUDE file for SYSGEN (for including driver)
- Chainfile for copying driver files for SYSGEN
- Memory allocation INCLUDE file (process control driver only)
- Memory allocation module (process control driver only)
- Switch file of modules in system
- Conditional file to bring in INCLUDE file for SYSGEN
- Chainfile for copying all files for SYSGEN
- Address offsets

F.2 EXAMPLES

F.2.1 Sample Assembly Chainfile

Filenames:
9992.&.xxxxDRV.AF (Process Control Drivers)
9993.&.xxxxDRV.AF (TRAP #2 drivers)

```
*****************************************************************************
*        9993.&.xxxxDRV.AF
****************************************************************************
=/*      &.RWINDRV.AF
=/* Chainfile to assemble the RWIN disk controller driver.
=/* If no output argument is specified for the listing, the
=/* chainfile will default to &.RWINDRV.LS =/* .
=/IFC 1
   =ARG &.RWINDRV.LS
=/ENDIF
=ASM &.RWINDRV.SA,&.RWINDRV.RO,\1;-WRZ=186
=/*{ Included files are:
=/*   9995.&.STR.EQ
=/*   9995.&.TCB.EQ
=/*   9995.&.CCB.EQ
=/*   9995.&.LV5.EQ
=/*   9995.&.IOE.EQ
=/*   9995.&.NIO.EQ
=/*   COMCMD.AI
=/*   RWININT.AI
=/*}
=END
```

F.2.2 Link File for SYSGEN

Filenames:
9992.&.xxxxDRV.LG
9993.&.xxxxDRV.LG

```
*****************************************************************
*
*        9992.&.xxxxDRV.LG
*
******************************************************************
=/*
=/*      &.M610DRV.LG
=/*
=/* Link chainfile run at SYSGEN time to link MVME610 driver.
=/*
=/* SYSGEN parameter LINKLS = \LINKLS = file/device to which to send
=/* the linker listing.
=/*
=/* SYSGEN parameter M610DRV = \M610DRV = address at which to link driver.
=/*
=LINK ,&.M610DRV.LO,\LINKLS;MIXH
SEG SEG0:0 \M610DRV
INPUT &.M610DRV.RO
INPUT &.SYSPAR.RO
END
=/*
=END
```

```
*****************************************************************
*
*        9993.&.xxxxDRV.LG
*
******************************************************************
=/*
=/*      &.RWINDRV.LG
=/*
=/* Link chainfile run at SYSGEN time to link RWIN1 driver.
=/*
=/* SYSGEN parameter LINKLS = \LINKLS = file/device to which to send
=/* the linker listing.
=/*
=/* SYSGEN parameter RWINDRV = \RWINDRV = address at which to link driver
=/*
=LINK ,&.RWINDRV.LO,\LINKLS;HAMIXS
SEGMENT RWIN:0-15 \RWINDRV
INPUT &.RWINDRV.RO
END
=/*
```

F.2.3 Configuration Files (DCBs and CDBs)

Filenames:

9992.&.xxxxIOC.SI	Process control drivers only
9992.&.IOCINT.AG	Process control drivers only
9993.IOC.xxxxDRV.AG	TRAP #2 drivers

```
**************************************************************************
*
*        9992.&.xxxxIOC.SI
*
**************************************************************************
*        INCLUDE  &.M610IOC.SI
*=========================================================================
*
* THIS IS THE CODE NEEDED TO SET UP CCBS FOR MVME610 or MVME620 MODULES.
*
* Add a CCB definition for each additional module in the system.
* Note that the memory mapped I/O address must be unique for each
* module, must not conflict with any other module in the system, and
* must match the onboard jumper selections. The I/O Channel interrupt
* vector number and hardware interrupt level must correspond to the
* interrupt jumper on the module.
*
*======================================================================
*
M610     EQU       \M610DRV
*
L610$01  SET       \L610$01
L610$02  SET       \L610$02
*
*        Set up 610/620 module CCBs (for two modules, in this example).
*
 CDB XOPEXC,'IN01',$80,0,M610,0,L610$01,3,\IOCVEC1,\IOCLVL1,$FF
&    0,0,0,0,0,0,0,0,0,0,0
*
 CDB XOPEXC,'IN02',$80,0,M610,0,L610$02,3,\IOCVEC2,\IOCLVL2,$FF,
&    0,0,0,0,0,0,0,0,0,0,0

*************************************************************************
*
*        9992.&.IOCINT.AG  modification
*
*************************************************************************
*   If any MVME610 or MVME620 modules are specified in <system>.CNFGDRVR.CI,
* include the code to perform CCB allocations for those modules.
*

  IFNE \NVME610+\NVME620
           INCLUDE   &.XM610IOC.SI
  ENDC
```

```
****************************************************************************
*
*        9993.&.xxxxDRV.AG
*
*        DISK EXAMPLE
*
******************************************************************************

         PAGE
*
*        IOC.RWINDRV.AG
*
*  Included equate files:
*      &.IOE.EQ
*      &.NIO.EQ
*
 NOLIST
  INCLUDE   &.IOE.EQ
  INCLUDE   &.NIO.EQ
 LIST

*  Included device-specific macros:
*      MACRO.DCB.SI
*      MACRO.DCBDISK.SI
*
 NOLIST
  INCLUDE MACRO.DCB.SI
  INCLUDE MACRO.DCBDISK.SI
 LIST

 PAGE
*  Define variables using SYSGEN parameters
*
RWIN     EQU \RWINDRV  address of RWIN1 driver
LWIN$01  EQU \LWIN$01  RWIN1 module #1 address
LWIN$02  EQU \LWIN$02  RWIN1 module #2 address
NHRWIN$1 EQU \NHRWIN$1 # hard disks on 1st RWIN1 controller module
NFRWIN$1 EQU \NFRWIN$1 # floppy disks on 1st RWIN1 controller module
NHRWIN$2 EQU \NHRWIN$2 # hard disks on 2nd RWIN1 controller module
NFRWIN$2 EQU \NFRWIN$2 # floppy disks on 2nd RWIN1 controller module

*******************************************************************************
*
*   SET UP DCBS FOR RWIN1
*******************************************************************************
*
*  Set up DCB parameters for RWIN1
*  (media independent)
*
DEV_ATT  SET       $1F
DEV_CODE SET       0
DEV_STAT SET       4
PAR_MASK SET       $1AF3
ECC_LEN  SET       0
```

```
***********************************************************************
*
*        HARD DISKS ON FIRST RWIN1 CONTROLLER
*
***********************************************************************

CHAN_ID  SET       'WIN1'
         IFGE      NHRWIN$1-1
***********************************************************************
*
*   First hard disk on first RWIN1 controller
*
***********************************************************************

DSKNM    SET       'HD\CONTWIN1\ZERO'
SIZESET  SET       0          If nonzero then a disk media has been selected.

         IFC       \RWIN0$1,'H8WIN10'
*
*  8" 10Mb hard disk
*
                   INCLUDE   &.H8WIN10.SI
SIZESET  SET       1
         ENDC
         IFC       \RWIN0$1,'H5WIN05'
*
*  5-1/4"  5Mb hard disk
*
                   INCLUDE   &.H5WIN05.SI
SIZESET  SET       1
         ENDC
         IFC       \RWIN0$1,'H5WIN10'
*
*  5-1/4" 10Mb hard disk
*
                   INCLUDE   &.H5WIN10.SI
SIZESET  SET       1
         ENDC
         IFC       \RWIN0$1,'H5WIN15'
*
*  5-1/4" 15Mb hard disk
*
                   INCLUDE   &.H5WIN15.SI
SIZESET  SET       1
         ENDC
         IFC       \RWIN0$1,'H5WIN40'
*
*  5-1/4" 40Mb hard disk
*
                   INCLUDE   &.H5WIN40.SI
SIZESET  SET       1
         ENDC
```

```
*
*   Make sure the user has defined a valid media type.
*
         IFEQ      SIZESET
         FAIL      ** Invalid media type
         ENDC
*
*  Set up attribute mask for hard disk on RWIN1
*
ATT_MASK SET  0<<IOADDEN+0<<IOATDEN+0<<IOADSIDE+0<<IOAFRMT
ATT_MASK SET  ATT_MASK+1<<IOARDISC+0<<IOADDEND+0<<IOATDEND+0<<IOARIBS
ATT_MASK SET  ATT_MASK+0<<IOADPCOM+0<<IOASIZE+1<<IOAALT

ATT_WORD SET  ATT_WORD+1<<IOAALT
*
*  Set up interleave for hard disk on RWIN1
*
INTERLEAVE SET     1

    DSKDCB   DSKNM,IOSID,IOSESS,DEV_ATT,DEV_CODE,DEV_STAT,CHAN_ID,
&            DSKNM&$F,ATT_MASK,PAR_MASK,ATT_WORD,\DCP$VSS,0,\DCP$WTO,
&            \DCP$RTO,SECT_PER_TRK,NUM_HEADS,CYL_DISK,INTERLEAVE,SPIRAL_OFF,
&            BYTES_PER_SECTOR,START_HEAD_NUM,CYL_DRIVE,PRE_COMP,SECT_DRIVE,
&            STEP_RATE,R_W_PRE_COMP,ECC_LEN
         ENDC


         IFGE      NHRWIN$1-2

********************************************************
*
*   Second hard disk on first RWIN1 controller
*
********************************************************

DSKNM    SET       DSKNM+1
SIZESET  SET       0         If nonzero then a disk media has been selected.

         IFC       \RWIN1$1,'H8WIN10'
*
*  8" 10Mb hard disk
*
                   INCLUDE   &.H8WIN10.SI
SIZESET  SET       1
         ENDC
         IFC       \RWIN1$1,'H5WIN05'
*
*  5-1/4" 5Mb hard disk
*
                   INCLUDE   &.H5WIN05.SI
SIZESET  SET       1
         ENDC
         IFC       \RWIN1$1,'H5WIN10'
```

```
*
*  5-1/4" 10Mb hard disk
*
                    INCLUDE   &.H5WIN10.SI
SIZESET  SET        1
         ENDC
         IFC        \RWIN1$1,'H5WIN15'
*
*  5-1/4"  15Mb hard disk
*
                    INCLUDE   &.H5WIN15.SI
SIZESET  SET        1
         ENDC
         IFC        \RWIN1$1,'H5WIN40'
*
*  5-1/4" 40Mb hard disk
*
                    INCLUDE   &.H5WIN40.SI
SIZESET  SET        1
         ENDC
*
*   Make sure the user has defined a valid media type.
*
         IFEQ       SIZESET
         FAIL       ** Invalid media type
         ENDC
*
*  Set up attribute mask for hard disk on RWIN1.
*
ATT_MASK SET  0<<IOADDEN+0<<IOATDEN+0<<IOADSIDE+0<<IOAFRMT
ATT_MASK SET  ATT_MASK+1<<IOARDISC+0<<IOADDEND+0<<IOATDEND+0<<IOARIBS
ATT_MASK SET  ATT_MASK+0<<IOADPCOM+0<<IOASIZE+1<<IOAALT

ATT_WORD SET  ATT_WORD+1<<IOAALT
*
*  Set up interleave for hard disk on RWIN1
*
INTERLEAVE SET      1

    DSKDCB   DSKNM,IOSID,IOSESS,DEV_ATT,DEV_CODE,DEV_STAT,CHAN_ID,
&            DSKNM&$F,ATT_MASK,PAR_MASK,ATT_WORD,\DCP$VSS,0,\DCP$WTO,
&            \DCP$RTO,SECT_PER_TRK,NUM_HEADS,CYL_DISK,INTERLEAVE,SPIRAL_OFF,
&            BYTES_PER_SECTOR,START_HEAD_NUM,CYL_DRIVE,PRE_COMP,SECT_DRIVE,
             STEP_RATE,R_W_PRE_COMP,ECC_LEN
         ENDC


**********************************************************************
*
*   FLOPPY DISKS ON FIRST RWIN1 CONTROLLER
*
**********************************************************************


         IFGE       NFRWIN$1-1
```

F

```
*************************************************************
*  First floppy disk on first RWIN1 controller
*************************************************************

DSKNM    SET      'FD\CONTWIN1\TWO'
SIZESET  SET      0         If nonzero then a disk media has been selected.

         IFC      \RWIN2$1,'F8DDDSI'
*
   8" double-density, double-sided, IBM format
*
                  INCLUDE    &.F8DDDSI.SI
SIZESET  SET      1
         ENDC
         IFC      \RWIN2$1,'F8SDDSM'
*
*  8" single-density, double-sided, Motorola format
*
                  INCLUDE    &.F8SDDSM.SI
SIZESET  SET      1
         ENDC
         IFC      \RWIN2$1,'F8SDSSI'
*
*  8" single-density, single-sided, IBM format
*
                  INCLUDE    &.F8SDSSI.SI
SIZESET  SET      1
         ENDC
         IFC      \RWIN2$1,'F8SDSSM'
*
*  8" single-density, single-sided, Motorola format
*
                  INCLUDE    &.F8SDSSM.SI
SIZESET  SET      1
         ENDC
         IFC      \RWIN2$1,'F5DDDSI'
*
*  5-1/4" double-density, double-sided, IBM format
*
                  INCLUDE    &.F5DDDSI.SI
SIZESET  SET      1
         ENDC
*   Make sure the user has defined a valid media type.
         IFEQ     SIZESET
         FAIL     ** Invalid media type
         ENDC
*
*  Set up attributes mask for floppy on RWIN1
*
ATT_MASK SET  1<<IOADDEN+1<<IOATDEN+1<<IOADSIDE+1<<IOAFRMT
ATT_MASK SET  ATT_MASK+1<<IOARDISC+0<<IOADDEND+0<<IOATDEND+0<<IOARIBS
ATT_MASK SET  ATT_MASK+0<<IOADPCOM+0<<IOASIZE+0<<IOAALT

ATT_WORD SET  ATT_WORD+0<<IOAALT
```

```
*
*  Set up interleave for floppy on RWIN1
*
INTERLEAVE SET     1

    DSKDCB   DSKNM,IOSID,IOSESS,DEV_ATT,DEV_CODE,DEV_STAT,CHAN_ID,
&            DSKNM&$F,ATT_MASK,PAR_MASK,ATT_WORD,\DCP$VSS,0,\DCP$WTO,
&            \DCP$RTO,SECT_PER_TRK,NUM_HEADS,CYL_DISK,INTERLEAVE,SPIRAL_OFF,
&            BYTES_PER_SECTOR,START_HEAD_NUM,CYL_DRIVE,PRE_COMP,SECT_DRIVE,
&            STEP_RATE,R_W_PRE_COMP,ECC_LEN
         ENDC

         IFGE      NFRWIN$1-2
***************************************************************
*
*  Second floppy disk on first RWIN1 controller
*
******************************************************************

DSKNM    SET       DSKNM+1
SIZESET  SET       0         If nonzero then a disk media has been selected.

         IFC       \RWIN3$1,'F8DDDSI'
*
*  8" double-density, double-sided, IBM format
*
                   INCLUDE    &.F8DDDSI.SI
SIZESET  SET       1
         ENDC
         IFC       \RWIN3$1,'F8SDDSM'
*
*  8" single-density, double-sided, Motorola format
*
                   INCLUDE    &.F8SDDSM.SI
SIZESET  SET       1
         ENDC
         IFC       \RWIN3$1,'F8SDSSI'
*
*  8" single-density, single-sided, IBM format
*
                   INCLUDE    &.F8SDSSI.SI
SIZESET  SET       1
         ENDC
         IFC       \RWIN3$1,'F8SDSSM'
*
*  8" single-density, single-sided, Motorola format
*
                   INCLUDE    &.F8SDSSM.SI
SIZESET  SET       1
         ENDC
         IFC       \RWIN3$1,'F5DDDSI'
```

```
*
*  5-1/4" double-density, double-sided, IBM format
*
                   INCLUDE    &.F5DDDSI.SI
SIZESET  SET       1
         ENDC
*   Make sure the user has defined a valid media type.
         IFEQ      SIZESET
         FAIL      ** Invalid media type
         ENDC
*
*  Set up attributes mask for floppy on RWIN1
*
ATT_MASK SET  1<<IOADDEN+1<<IOATDEN+1<<IOADSIDE+1<<IOAFRMT
ATT_MASK SET  ATT_MASK+1<<IOARDISC+0<<IOADDEND+0<<IOATDEND+0<<IOARIBS
ATT_MASK SET  ATT_MASK+0<<IOADPCOM+0<<IOASIZE+0<<IOAALT

ATT_WORD SET  ATT_WORD+0<<IOAALT
*
*  Set up interleave for floppy on RWIN1
*
INTERLEAVE SET     1

    DSKDCB   DSKNM,IOSID,IOSESS,DEV_ATT,DEV_CODE,DEV_STAT,CHAN_ID,
&            DSKNM&$F,ATT_MASK,PAR_MASK,ATT_WORD,\DCP$VSS,0,\DCP$WTO,
&            \DCP$RTO,SECT_PER_TRK,NUM_HEADS,CYL_DISK,INTERLEAVE,SPIRAL_OFF,
&            BYTES_PER_SECTOR,START_HEAD_NUM,CYL_DRIVE,PRE_COMP,SECT_DRIVE,
&            STEP_RATE,R_W_PRE_COMP,ECC_LEN
         ENDC

*******************************************************************************
*
*   SET UP CDBS FOR FIRST RWIN1
*******************************************************************************
*

***  CHANNEL DATA BLOCK  ***
    CDB     0,CHAN_ID,XTDWIN,254,RWIN,0,LWIN$01,1,\IOCVEC3,\IOCLVL3,$10,0,0, &
         0,0,0,0,0,0,0,0,0

*
*
*  OTHER RWIN MODULE DEVICES ARE SET UP SIMILARLY
*
*


         END
*****************************************************************************
*
*        9993.&.xxxxDRV.AG
*
*        TERMINAL EXAMPLE
*
******************************************************************************
```

```
*
*        IOC.MPSCDRV.AG
*
*   Included equate files:
*       &.IOE.EQ
*       &.NIO.EQ
*       &.LV5.EQ
*
         NOLIST
         INCLUDE   &.IOE.EQ
         INCLUDE   &.NIO.EQ
         INCLUDE   &.LV5.EQ
         LIST

*   Included device-specific macros
*       MACRO.DCB.SI
*       MACRO.DCBTERM.SI
*
         NOLIST
         INCLUDE   MACRO.DCB.SI
         INCLUDE   MACRO.DCBTERM.SI
         LIST

         INCLUDE   \&FILENAM

         PAGE
*
*   Assign values from SYSGEN parameters
*
DVCODE     SET \&CRTDV    Define starting device code

*   Define the physical addresses of the drivers
MPSC       EQU \MPSCDRV   common 7201 driver
SDRVADD    EQU \&SDRVADD

Define offset from Port A to Port B (hardware constant on MVME400 module)
PORTB EQU 1*\CMULT

*   Define TCP$ATW one time here for all terminals
TCP$ATW   SET  \TCP$HCPY+\TCP$XCTL<<1+\TCP$BITS<<2+\TCP$STPB<<3+\TCP$USEP<<4
TCP$ATW   SET  TCP$ATW+\TCP$PRTY<<5+\TCP$ECHO<<6+\TCP$TAHD<<7+\TCP$TFUL<<8
TCP$ATW   SET  TCP$ATW+\TCP$PNUL<<9+\TCP$MODM<<10+\TCP$OFFH<<11

******************************************************************************
*
*   SET UP DCB FOR FIRST MPSC PORT ON FIRST MODULE
*
******************************************************************************
*****  1st MPSC port  *****
*
 IFGE NUBRD1-1
*    1st MPSC port on 1st module
*
SNAME    SET '\&SDRVR'<<8+(DVCODE&$FF)  supervisor channel name mnemonic
```

```
CNAME    SET 'CMP'+(DVCODE&$FF)  channel name mnemonic
*
*-- DATA CONTROL BLOCK  --*
 CRTDCB DVCODE,IOSID,IOSESS,$133,35,1,CNAME,0,ATT_MASK,PAR_MASK,TCP$ATW,
&   \TCP$REC,\TCP$RSZ,\TCP$WTO,\TCP$RTO,\TCP$XOF,\TCP$XON,\TCP$BRC,\TCP$DOP,
&   \TCP$RLN,\TCP$CLC,\TCP$RTV,\TCP$EOL,\TCP$BRT,\TCP$NLS,\TCP$TRC,\TCP$TTP

***  CHANNEL DATA BLOCK  ***
 CDB $0011,SNAME,0,254,SDRVADD,0,DEVADDR1,1,IOCVECT,IOCLVL,
&   $30,0,0,0,0,0,0,0,0,0,0,0
 CDB $0009,CNAME,CHANTYP1,254,MPSC,SNAME,DEVADDR1,1,IOCVECT,IOCLVL,
&   $30,0,0,0,0,0,0,0,0,0,0,0
DVCODE    SET  DVCODE+1
 ENDC

*
*  OTHER PORTS ARE SET UP SIMILARLY ....
*


 END
***************************************************************************
*
*        9993.&.xxxxDRV.AG
*
*        PRINTER EXAMPLE
******************************************************************************

*        IOC.PIADRV.AG
*
*   Included equate files:
*        &.IOE.EQ
*        &.NIO.EQ
*        &.LV5.EQ

         NOLIST
         INCLUDE   &.IOE.EQ
         INCLUDE   &.NIO.EQ
         INCLUDE   &.LV5.EQ
         LIST

*   Included device-specific macros
*        MACRO.DCB.SI
*        MACRO.DCBPRT.SI

         NOLIST
         INCLUDE   MACRO.DCB.SI
         INCLUDE   MACRO.DCBPRT.SI
         LIST

         INCLUDE   \&FILENAM          Include file for specific PIA
*
*   Assign values from SYSGEN parameters
*
```

```
DVCODE   SET       \&PRTDV   Define starting device code
PIA      EQU       \PIADRV   Define the physical address of the driver

*   Define offset from Port A to Port B (hardware constant on MVME410 module)
*
PORTB  EQU       8

CNAME    SET       ('C'<<16)+(DVCODE>>8)  channel name mnemonic

*   Define printer attribute word (PCP$ATW) one time for all printers
*
PCP$ATW  SET       \PCP$TLRL<<2+\PCP$AFF<<1+\PCP$LNFD

*******************************************************************************
*
*  SET UP DCB FOR 1ST PIA (PORT A) PRINTER
*
*******************************************************************************

 IFGE  NUBRD1-1
*
*   PIA #1, printer port A
*
*--  DATA CONTROL BLOCK  --*
*
 PRTDCB DVCODE,IOSID,IOSESS,$632,95,1,CNAME,0,$0007,$0033,PCP$ATW,\PCP$REC,
&       \PCP$RSZ,\PCP$WTO,\PCP$LRL,\PCP$ELC

 IFNE DVCODE-'PR  '
DVCODE SET DVCODE+$100  Set up for next printer ID
 ENDC
 IFEQ DVCODE-'PR  '
DVCODE SET 'PR1 '
 ENDC

*******************************************************************************
*
*  SET UP CDB FOR 1ST PIA (PORT A) PRINTER
*
*******************************************************************************

***  CHANNEL DATA BLOCK  ***
 CDB 0,CNAME,XTPRTL,254,PIA,0,DEVADDR1,7,IOCVECT,IOCLVL,$10,
&    0,0,0,0,0,0,0,0,0,0,0
 ENDC

*
*  OTHER PORTS ARE SET UP SIMILARLY
*

 END
```

F.2.4 Patch Chainfile

Filename:
9998.VERSAPT.xxxxDRV.CF

```
*********************************************************************
*
*        9998.VERSAPT.xxxxDRV.CF
*********************************************************************
=/*
=/* VERSAPT.RWINDRV.CF
=/*
=PATCH VERSADOS.SY
*>-------------------------------------------------------------------
0 \RWINDRV              * Start of RWINDRV
*<-------------------------------------------------------------------
QUIT
```

F.2.5 INCLUDE File for SYSGEN (for Including Driver)

Filenames:

9992.&.xxxxDRV.CI	Process Control Drivers
9993.&.xxxxDRV.CI	TRAP #2 drivers

```
************************************************************************
*
*        9992.&.xxxxDRV.CI
************************************************************************
*
*        &.M610DRV.CI
*
MSG    ******************************************************************
MSG    **  MVME610/620 DRIVER - AC INPUT/DC INPUT
MSG    ******************************************************************

*  Set Process Control Flag= ON
&PCDRV    = \&PCDRV+1

M610QSIZ  = 128          Minimum number of entries in Interrupt Processing
*                        Queue

* Build VERSAdos patch chainfile <system>.VERSAPT.CF
=COPY        VERSAPT.M610DRV.CF,VERSAPT.CF;A

M610DRV   = *            MVME610/620 driver base address
SUBS        &.M610IOC.SI CCB allocations
SUBS        &.M610DRV.LG Driver module LINK command file
LINK        &.M610DRV.LG
IFEQ        \LINKLSW
   =COPY       \LINKLS,\WORKLS;A
ENDC
PROCESS     &.M610DRV.LO
END         &.M610DRV.LO
```

```
*************************************************************************
*
*        9993.&.xxxxDRV.CI
*
****************************************************************************
*
*        &.RWINDRV.CI
*
MSG      *********************************************************************
MSG      **     RWIN1 DRIVER - Winchester Disk Controller
MSG      *********************************************************************

*  Adjust total number of disks -
&TOTDSK    = \&TOTDSK+\HRWIN$01+\FRWIN$01+\HRWIN$02+\FRWIN$02

* Build VERSAdos patch chainfile <system>.VERSAPT.CF
=COPY         VERSAPT.RWINDRV.CF,VERSAPT.CF;A

RWINDRV    = *
SUBS          &.RWINDRV.LG
LINK          &.RWINDRV.LG
IFEQ          \LINKLSW
   =COPY         \LINKLS,\WORKLS;A
ENDC
PROCESS       &.RWINDRV.LO
END           &.RWINDRV.LO

* Assemble the IOC portion and merge with current IOC.RO
SUBS          IOC.RWINDRV.AG
ASM           IOC.RWINDRV.AG,NEW.RO,\ASMLS;RMZ=100
IFEQ          \ASMLSW
   =COPY         \ASMLS,IOC.LIST.TF;A
ENDC

* Generate merged IOC.RO by appending new .RO file
INCLUDE       &.IOCGEN.CI
IFEQ          \LINKLSW
   =COPY         \LINKLS,IOC.LIST.TF;A
ENDC
```

F.2.6 Chainfile for Copying Driver Files for SYSGEN

Filename:
9998.COPYGEN.xxxxDRV.CF

```
*************************************************************************
*
*        9998.COPYGEN.xxxxDRV.CF
*
*        Process control driver
*
*************************************************************************
=/*
=/* COPYGEN.M610DRV.CF - Chainfile to copy all files for SYSGEN of MVME610
=COPY \1:9992.&.M610DRV.LG             \2:\3;\4C
=COPY \1:9992.&.M610DRV.RO             \2:\3;\4C
=COPY \1:9992.&.M610IOC.SI             \2:\3;\4C
=COPY \1:9992.&.M610MEM.AG             \2:\3;\4C
=COPY \1:9992.&.M610MEM.CI             \2:\3;\4C
=COPY \1:9992.&.M610DRV.CI             \2:\3;\4C
=COPY \1:9998.VERSAPT.M610DRV.CF       \2:\3;\4C


*************************************************************************
*
*        9998.COPYGEN.xxxxDRV.CF
*
*        TRAP #2 driver
*
*************************************************************************
=/*
=/* COPYGEN.RWINDRV.CF - Chainfile to copy all files for SYSGEN of RWIN1
=/*                       (Remote Winchester Driver)
=COPY \1:9993.&.RWINDRV.CI             \2:\3;\4C
=COPY \1:9993.&.RWINDRV.LG             \2:\3;\4C
=COPY \1:9993.&.RWINDRV.RO             \2:\3;\4C
=COPY \1:9998.IOC.RWINDRV.AG           \2:\3;\4C
=COPY \1:9998.VERSAPT.RWINDRV.CF       \2:\3;\4C
```

F.2.7 Memory Allocation INCLUDE File (Process Control Driver Only)

Filename:
9992.&.xxxxMEM.CI

```
*************************************************************************
*
*        9992.&.xxxxMEM.CI
*
*************************************************************************
*
*        &.M610MEM.CI
*
MSG         ****************************************************************
MSG         *  MVME610/620    driver initialization
MSG         ****************************************************************

SUBS        &.M610MEM.AG
ASM         &.M610MEM.AG,NEW.RO,\ASMLS
IFEQ        \ASMLSW
   =COPY       \ASMLS,\WORKLS;A
ENDC

INCLUDE     &.ROGEN.CI
```

F.2.8 Memory Allocation Module (Process Control Driver Only)

Filename:
9992.&.xxxxMEM.AG

```
*************************************************************************
*
*        9992.&.xxxxMEM.AG
*
*************************************************************************

*
*        &.M610MEM.AG
*
    IFNE   \NVME610+\NVME620
          NOFORMAT
          PAGE
PAGAL     EQU    4              RMS68K PAGAL TRAP #0 directive number
KILLER    EQU    32             RMS68K KILLER TRAP #0 directive number
          OPT   CRE

******************************************************************************
*
```

```
*           Interrupt Processing Queue Allocation and Initialization
*                for the Input Module (MVME610/620) Driver
*
*   This code section allocates memory for the Interrupt Queue Control Table
* and the Interrupt Processing Queue used by the Input Module Driver. First a
* page of memory is allocated for the control table, and the base address of
* the table is stored in SYSPAR's INPTBL field for later retrieval by the
* driver's initialization routine. The table is initialized and the memory
* required for the Interrupt Processing Queue is calculated and allocated
* based on SYSGEN parameter M610QSIZ. If all of the required pages of memory
* cannot be allocated, RMS68K routine KILLER is called to force system crash.
*   This code is linked into the System Initializer as described in the SYSGEN
* instructions for the driver. In the following code, symbols preceded by a
* backslash (\) character are user-specified SYSGEN parameters that are
* replaced with numeric values by the SYSGEN SUBS command prior to assembly
* of this source. The parameters are assumed to have been validated in a
* separate assembly performed earlier in the SYSGEN process.
*****************************************************************************
          SPC    2
          XREF   INPTBL             This symbol is an address within SYSPAR.
          XDEF   M610MEM
          SPC    2
          SECTION 8
          SPC    2
M610QSIZ  EQU    \M610QSIZ          Interrupt Processing Queue size (maximum
*                                   number of entries the queue is to contain).
          SPC    2
          INCLUDE  &.BAB.EQ
*         INCLUDE  &.BAB.EQ
          INCLUDE  &.M610INTQ.EQ
*         INCLUDE  &.M610INTQ.EQ
          SECTION  8
          SPC    2
M610MEM  EQU      *
INPMEM    CLR.L  D0                 Clear all registers to be used in calling
          MOVE.L D0,A0              the RMS68K memory allocation routine.
          MOVE.L D0,A1
          MOVE.L D0,A6
          SPC    2
          ADD.W  #1,A0              Call RMS68K routine PAGAL to allocate 256
          MOVE.B #PAGAL,D0          bytes (1 page) of memory for global vari-
          TRAP   #0                 able storage. If no memory is available,
          BRA.S  INPM010            call RMS68K routine KILLER to crash the
          BSR.S  INPMKILL           system.
          SPC    2
INPM010   MOVE.L A0,INPTBL          Save global variable area address returned
          MOVE.L A0,A4              by PAGAL in the input driver table pointer
*                                   (INPTBL) within SYSPAR.  Make a permanent
*                                   copy of the address in A3.
          SPC    2
          MOVE.W #63,D0             Clear the global variable area to all
INPM020   CLR.L  (A0)+              zeros.
          DBRA   D0,INPM020
          PAGE
```

```
* The following code uses the Interrupt Queue Control Table (IQCT) and
* allocates memory for the queue.
          SPC     2
INPM050   CLR.L   D1                  Clear D1 for use as work register.
          SPC     2
          MOVE.W #M610QSIZ,D5         Load the number of entries that the
*                                     Interrupt Queue is to contain into D5.
          SPC     2
          MOVE.L #'IQCT',QID(A4)      Initialize ASCII ID field in the IQCT with
*                                     the mnemonic IQCT. This serves as a visual
*                                     aid in analyzing memory dumps for debug.
          SPC     2
          MULU    #IQENTLN,D5         Calculate queue length in bytes by multi-
*                                     plying the number of entries in the queue
*                                     by the queue entry length.
          SPC     2
          TST.B   D5                  Calculate the minimum number of 256-byte
          BEQ.S   INPM055             memory pages necessary to accommodate the
          ADD.W   #256,D5             specified number of queue entries.
INPM055   LSR.W   #8,D5
          SPC     2
          MOVE.W #15,D0               Calculate bit number of the most signifi-
INPM057   LSL.W   #1,D5               cant bit of the number of pages to allo-
          DBCS    D0,INPM057          cate in D5.
          SPC     2
          BEQ.S   INPM058             If bits other than the most significant bit
          ADD.W   #1,D0               of the number of pages are set, increment
*                                     the number of the most significant bit to
*                                     the next higher power of two.
          SPC     2
INPM058   CLR.L   D5                  Set the number of pages to allocate to
          BSET    D0,D5               the lowest power of two that will hold the
*                                     requested number of queue entries.
          SPC     2
INPM060   MOVE.L D5,A0                Load the number of pages to be allocated
          MOVE.L #PAGAL,D0            into A0 and call RMS68K routine PAGAL to
          TRAP    #0                  obtain the memory. If the required amount
          BRA.S   INPM070             of memory is not available, call RMS68K
INPMKILL  MOVE.L #KILLER,D0           routine KILLER to crash the system.
          TRAP    #0
          SPC     2
INPM070   MOVE.L A0,QPTR(A4)          Store the Interrupt Processing Queue
*                                     base address in the IQCT.
          SPC     2
          ASL.W   #8,D5               Convert the length of the Interrupt Queue
          SUB.W   #1,D5               in 256-byte pages to the length in bytes.
          MOVE.W D5,QWAMSK(A4)        Form the queue wraparound mask by decre-
*                                     menting the queue length, and store the
*                                     mask in the queue control entry.
          SPC     2
*   This is the end of the Input Module Driver system initialization routine.
          FORMAT
    ENDC
         END
```

F.2.9 Switch File of Module in System

Filename:
9998.<system>.CNFGDRVR.CI

```
*****************************************************************************
*
*        9998.<system>.CNFGDRVR.CI  modification *
*        Process control driver example
*
*****************************************************************************

*----------------------------------------------------------------------------
NVME610   = 0           # of MVME610 AC input controller boards

*----------------------------------------------------------------------------
*****************************************************************************
*
*        9998.<system>.CNFGDRVR.CI  modification
*
*        TRAP #2 disk driver example *
*****************************************************************************
*----------------------------------------------------------------------------
NORWIN1   = 2                # of RWIN1 Winchester controller boards
IFGT         \NORWIN1
   CONTRWIN  = "0"           RWIN1 is controller 0
   HRWIN$01  = 2             # of hard   disk   drives on RWIN1; max= 2
   FRWIN$01  = 2             # of 5-1/4" floppy drives on RWIN1; max= 2

   RWIN0$01  = "'H5WIN15'"   Type of 1st hard   disk on 1st RWIN1, drive 0
   RWIN1$01  = "'H5WIN15'"   Type of 2nd hard   disk on 1st RWIN1, drive 1
   RWIN2$01  = "'F5DDDSI'"   Type of 1st floppy disk on 1st RWIN1, drive 2
   RWIN3$01  = "'F5DDDSI'"   Type of 2nd floppy disk on 1st RWIN1, drive 3
ENDC
IFGT       \NORWIN1-1
   CONTWIN2 = "1"           RWIN1 is controller 0
   HRWIN$02  = 2            # of hard   disk   drives on RWIN1; max= 2
   FRWIN$02  = 2            # of 5-1/4" floppy drives on RWIN1; max= 2

   RWIN0$02  = "'H5WIN15'"   Type of 1st hard   disk on 1st RWIN1, drive 0
   RWIN1$02  = "'H5WIN15'"   Type of 2nd hard   disk on 1st RWIN1, drive 1
   RWIN2$02  = "'F5DDDSI'"   Type of 1st floppy disk on 1st RWIN1, drive 2
   RWIN3$02  = "'F5DDDSI'"   Type of 2nd floppy disk on 1st RWIN1, drive 3
   *
   *     NOTE:  Do not mix 5-1/4" and 8" floppies.  Choose one or the other.
   *
ENDC

*----------------------------------------------------------------------------
```

F.2.10 Conditional File to Bring in INCLUDE File for SYSGEN

```
Filename:
9998.<system>.IFDRVR.CI
```

```
**********************************************************************************
*
*        9998.<system>.IFDRVR.CI       modification
*
**********************************************************************************
IFNE        \NVME610+\NVME620
   INCLUDE     &.M610DRV.CI
ENDC
*
*
IFNE        \NORWIN1
   INCLUDE     &.RWINDRV.CI
ENDC
*
```

F.2.11 Chainfile For Copying All Files For SYSGEN

```
Filename:
9998.<system>.COPYSGEN.CF
```

```
**********************************************************************************
*
*        9998.<system>.COPYSGEN.CF  modification
*
**********************************************************************************
=/*-------------------------------------------------------------------------
=/* Copy all VME610 files for SYSGEN
=/@    \1:9998.COPYGEN.M610DRV.CF
=/*-------------------------------------------------------------------------
=/*-------------------------------------------------------------------------
=/* Copy all RWIN1 (Winchester) driver files for SYSGEN
=/@    \1:9998.COPYGEN.RWINDRV.CF
=/*-------------------------------------------------------------------------
```

F.2.12 Address Offsets

Filenames:

IOC.ADDRESS.CI	For modules on I/O Channel
SIO.ADDRESS.CI	For modules in short address space

```
*******************************************************************************
*
*        IOC.ADDRESS.CI
*
*******************************************************************************
*  I/O Channel Address Offsets for Standard SYSGENs
*******************************************************************************
*
*  NOTES:  1. These addresses are coded into many of the firmware debuggers
*             so the bug can access the device.  REQUIRED FOR BOOTING!
*          2. A maximum of 16 devices can be on the channel at one time
*             for guaranteed operation in a "worst case" situation.
*
*  Symbol format is:    Lnnn$mm
*    where:     L= location
*             nnn= VMEmodule number/description or Vnn for VERSAmodule
*               $= separator
*              mm= module number 01-99
*
*   "CMULT" is defined in the "<system>.SYSTEM.CI" file.
*   "+$NN"  is the offset to the 1st module register.
*-------------------------------------------------------------------------
*-------------------------------------------------------------------------
L610$01   = \IOCBASE+($002*\CMULT)+$1  MVME610 module #1, port address
*                                         L610$01  = \L610$01
L610$02   = \IOCBASE+($004*\CMULT)+$1  MVME610 module #2, port address
*                                         L610$02  = \L610$02
*-------------------------------------------------------------------------
*-------------------------------------------------------------------------
LWIN$01   = \IOCBASE+($068*\CMULT)+$03 RWIN1   module #1, port address
*                                         LWIN$01  = \LWIN$01
LWIN$02   = \IOCBASE+($060*\CMULT)+$03 RWIN1   module #2, port address
*                                         LWIN$02  = \LWIN$02
*------------------------------------------------------------------------------
*******************************************************************************
*
*        SIO.ADDRESS.CI
*
*******************************************************************************
*  Short I/O Address Space Offsets for Standard SYSGENs
*******************************************************************************
*
*  NOTE:  1. These addresses are coded into many of the firmware debuggers
*            so the bug can access the device.  REQUIRED FOR BOOTING!
*
*
```

```
*
*  Symbol format is:    Lnnn$mm
*    where:    L= location
*            nnn= VMEmodule number/description or Vnn for VERSAmodule
*              $= separator
*             mm= module number 01-99
*
*------------------------------------------------------------------------
L320$01   = \SIOBASE+$B00D VME320 module #1 address
*                               L320$01  = \L320$01
L320$02   = \SIOBASE+$B01D VME320 module #2 address
*                               L320$02  = \L320$02
*------------------------------------------------------------------------
```

F.2.13 Conditional File to Bring in Memory Allocation File (Process Control Drivers Only)

Filename:
9992.&.PCDRV.CI

```
****************************************************************************
*
*         9992.&.PCDRV.CI  modification
*
*************************************************************************
*
 IFGT        \NVME610+\NVME620
   INCLUDE     &.M610MEM.CI
 ENDC
*
```

THIS PAGE INTENTIONALLY LEFT BLANK.

APPENDIX G

BACKGROUND AND CALL-GUARDED MODES

G.1 INTRODUCTION

This appendix describes the background and call-guarded modes of execution available to code running in supervisor mode.

G.2 BACKGROUND MODE

An Interrupt Service Routine (ISR) that runs in supervisor mode may not need to do all of its work at interrupt level. After a few instructions, the remaining work can often be done at interrupt level 0, allowing more timely service of other interrupts that are pending.

However, an ISR cannot simply lower its mask to 0. It might have interrupted a lower level interrupt service routine, which might not be ready to process another interrupt.

What an ISR can do is schedule a piece of code to be run in background mode (i.e., at interrupt level 0 after all the ISRs currently servicing interrupts have exited). The RMS68K subroutine BKG_SCHEDULE, also available from TRAP #0, performs the scheduling (refer to paragraph G.4). When the interrupt level returns to 0, scheduled background service routines are executed.

Figure 1 illustrates the execution path when several interrupts are taken that schedule background activity. Notice when, and in what order, the background routines are executed (dispatched).

Because background routines are dispatched at interrupt level 0, they do not block other pending interrupts. The routines run in supervisor mode, as do the drivers, and thus have access to all the system resources. Scheduling and dispatch of background routines is intended to be very efficient and does not provide frills. Routines execute in the order in which they were scheduled; there is no timeslicing of background routines, nor any prioritization.

When a background routine is running and another interrupt occurs that also schedules background activity, the new background work is not started immediately. Instead, execution of the already started background routine is resumed. Only after all previously scheduled background routines have run does the newly scheduled routine run. This can be thought of as a sort of mutual exclusion property of background execution.

This is a very useful feature for driver writers. A driver can have several background routines that access a single data structure, yet each can do so without fear of corruption by other background routines because they cannot possibly run until this one is done. Contrast this to ordinary interrupt processing, where only by keeping interrupts masked can a routine be assured that one of the other routines does not run as the result of an interrupt and corrupt the data structure.

```
==============================================================================
  normal user
 or supervisor
   mode code
      |
       ------------------------------------
                                           |
                                     start level 2
                                     interrupt and
                                   schedule routine A
                                           |
                                            ---------------------------
                                                                      |
                                                              process level 4
                                                               interrupt and
                                                            schedule routine B
                                                                      |
                                            --------------------------
                                           |
                                   finish level 2
                                   interrupt and
                                  schedule routine C
                                           |
                         -----------------
                        |
                dispatch background
                 routine A, then B
                        |
                         -----------------
                                           |
                                   dispatch level 3
                                    interrupt and
                                   schedule routine D
                                           |
                        ------------------
                       |
            background routine B
                finishes; C is
              dispatched, then D
                       |
          -------------
         |
 resume normal user
   or supervisor
    mode code
==============================================================================
```

FIGURE 1. Interrupt Execution Path

Scheduling a routine for execution in background entails setting a data structure called a Background Activation Block (BAB) to a queue of BABs called the background queue. The BAB structure is illustrated in Figure 7-1.

System equate file 9995.&.BAB.EQ defines this structure. The BABUSE flag should be initialized to zero (false) by the driver before its first use; thereafter it will be maintained by the scheduler and dispatcher and should not be touched by the driver. Usually the BAB is part of a larger data structure, such as a CCB and, in fact, the background routine uses that knowledge to find the data structure.

The background queue is initialized and maintained entirely by RMS68K. Its structure appears in Figure 2.

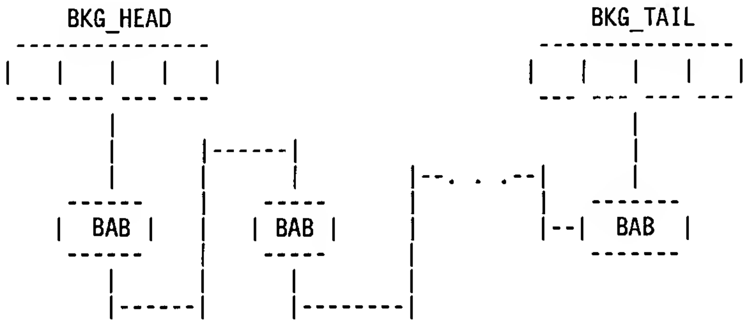


FIGURE 2. Background Queue Structure

If the queue is not empty, the head pointer points to the first BAB in the queue and the tail pointer points to the last. The BABLNK field of each BAB points to the next BAB, except that the BABLNK of the last BAB is zero.

If the queue is empty, the tail pointer points to the head pointer and the head pointer is zero.

NOTE

> All ISRs must exit through the common interrupt handler. This is important even for those routines that do not use the background mechanism, because failure to do so might result in failure to dispatch some scheduled background activity before returning to normal user or supervisor mode. This is handled automatically by drivers under the CMR software.

G.3 CALL-GUARDED MODE

A routine that is not running as the result of an interrupt may require the mutual exclusion relationship with background routines. In other words, a routine may be written that will be called during initialization or command service processing which uses data structures that are also used by background routines. Ordinary code can be interrupted at any time, and background routines can run as a result.

To protect noninterrupt routines, they should be called using the CALL_GUARDED mechanism rather than a BSR or JSR instruction. Figure 3 shows how this protection works.

```
==============================================================================

             normal supervisor
                 mode code
                     |
                     |------------|
                                  |
                            begin guarded
                              routine A
                                  |---------------|
                                                  |
                                            process level 3
                                             interrupt and
                                           schedule routine B
                                                  |
                                  |---------------|
                                  |
                          complete guarded
                        routine A; dispatch
                        background routine B
                                  |
                     |------------|
                     |
               resume normal
                supervisor
                mode code
==============================================================================
```

FIGURE 3. Call-Guarded Mode Path

G.4 BKG_SCHEDULE

BKG_SCHEDULE schedules a routine for execution in background mode (i.e., it puts a BAB on the background queue). When the BKGRND and SET_BAB macros described in Chapter 7 ("Background Routines Called With the BKGRND Macro") are used, the BKG_SCHEDULE macro is not required (its function has been accomplished).

If the call is made using TRAP #0, register D0 contains the service request code. A0 contains the address of the BAB representing the routine to be scheduled for background execution.

All registers are preserved by the call. There is only one return from the call: successful completion. If the BAB was not already on the background queue, it has been added; if it was already on the queue, it has not been added again.

On entry to the background routine, the contents of registers D0 through D7 and A0 through A6 have been saved and can be used. A1 points to the BAB that was used to schedule the routine. The interrupt level is 0 and supervisor mode is set.

When the background routine exits, supervisor mode must be set and the interrupt level must be 0. Exit is accomplished with an RTS instruction, so on exit the stack must be restored to its entry value. Except for A7, registers do not have to be preserved.

The calling sequence using TRAP #0 is

```
        LEA      <BAB address>,A0
        MOVE.L   #36,D0
        TRAP     #0
```

The calling sequence as a subroutine call is

```
        LEA      <BAB address>,A0
        BSR      BKG_SCHEDULE
```

G.5 CALL_GUARDED

CALL_GUARDED executes a routine in guarded mode so that it is guaranteed non-reentrant execution in relation to background processing. This call should be made only from interrupt level 0, and from code that is not running as the result of an interrupt.

If the call is made using TRAP #0, register D0 contains the service request code. A0 always contains the address of the routine to be executed in guarded mode. Registers D1 through D7 and A1 through A6 (and D0 if the call is made as a subroutine) can be used to pass parameters to the guarded routine.

On entry to the guarded mode routine, A0 contains garbage; D0 also contains garbage if the call is made with TRAP #0. All other registers except A7 are as the calling routine left them. The interrupt level is 0 and supervisor mode is set.

When the guarded routine exits, supervisor mode must be set and the interrupt level must be 0. Exit is accomplished with an RTS instruction, so on exit the stack must be restored to its entry value. Only register A7 must be preserved, so any other registers can be used to pass information back to the calling routine.

The calling sequence using TRAP #0 is

```
        LEA      <address of guarded routine>,A0
        BSR      CALL_GUARDED
```

The calling sequence as a subroutine call is

```
        LEA      <address of guarded routine>,A0
        MOVE,L   #37,D0
        TRAP     #0
```

G.6 ENTRY

ENTRY is a routine which allows a routine that was not linked with RMS68K to find the addresses of certain RMS68K routines.

Software programs that run in supervisor mode but are not linked with RMS68K have access to certain RMS68K features through the TRAP #0 instruction. When a TRAP #0 is performed, the value in D0 identifies the routine to be called.

Sometimes even the low overhead of a TRAP #0 is undesirable. In these cases, users can obtain the actual address of the routine using the ENTRY call during initialization, and then just call the module as a subroutine. ENTRY can also be used to verify the existence of a particular TRAP #0 directive.

When calling ENTRY using TRAP #0 (it would make no sense to call it as a subroutine, because that implies that the routine is linked with the Executive), register D0 contains the service request code of the ENTRY routine. Register D1 contains the service request code of the routine whose address is desired.

If a routine exists for the specified code (i.e., D1 is not out of range), return is to the point immediately following the TRAP #0. A0 contains the address of the specified routine.

If no such routine exists, return is to the position immediately following the TRAP #0 plus two bytes.

The ENTRY call structure is as follows:

```
           MOVE.L   #<directive # whose address is desired>,D1
           MOVE.L   #35,D0                Call the ENTRY routine to find the
                                          address of this routine
           TRAP     #0
           BRA.S    EXISTS                Good return
           BRA      DOES-NOT-EXIST        Bad return

EXISTS     MOVE.L   A0,RTN ADDR           Save address returned from ENTRY call
             .
             .
             .
           MOVE.L   RTN SFFT.A3           A3 is address of the RMS68K routine;
                                          call the RMS68K routine directly
```

SUGGESTION/PROBLEM REPORT

MICROSYSTEMS
QUALITY • PEOPLE • PERFORMANCE

Motorola welcomes your comments on its products and publications. Please use this form.

To: Motorola Inc.
Microsystems
2900 S. Diablo Way
Tempe, Arizona 85282
Attention: Publications Manager
Maildrop DW164

Product:

Manual:

COMMENTS:

Please Print

Name

Title

Company

Division

Street

Mail Drop

Phone

City

State

Zip

For Additional Motorola Publications
Literature Distribution Center
616 West 24th Street
Tempe, AZ 85282
(602) 994-6561

Four Phase/Motorola Customer Support, Tempe Operations
(800) 528-1908
(602) 438-3100

MOTOROLA

MOTOROLA *Semiconductor Products Inc.*
P.O. BOX 20912 • PHOENIX, ARIZONA 85036 • A SUBSIDIARY OF MOTOROLA INC.

19134 PRINTED IN USA (1/86) MESSENGER 1500